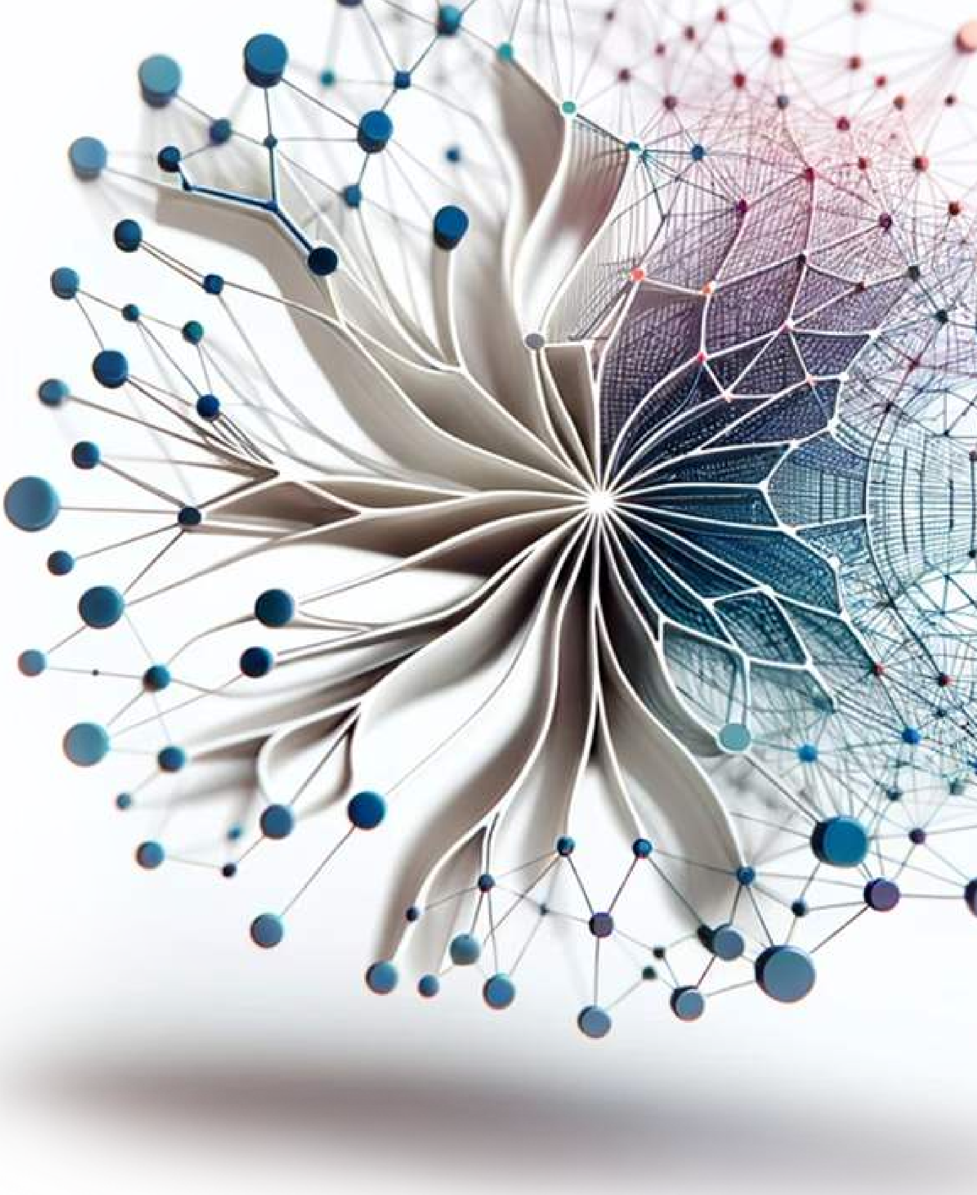


Introduction to

Machine Learning Systems

Vijay
Janapa Reddi

Machine Learning Systems

Principles and Practices of Engineering Artificially Intelligent Systems

Prof. Vijay Janapa Reddi
School of Engineering and Applied Sciences
Harvard University

With heartfelt gratitude to the community for their invaluable contributions and steadfast support.

June 10, 2025

Table of contents

Preface

Welcome to Machine Learning Systems, your gateway to the fast-paced world of machine learning (ML) systems. This book is an extension of the CS249r course at Harvard University, taught by Prof. Vijay Janapa Reddi, and is the result of a collaborative effort involving students, professionals, and the broader community of AI practitioners.

We've created this open-source book to demystify the process of building efficient and scalable ML systems. Our goal is to provide a comprehensive guide that covers the principles, practices, and challenges of developing robust ML pipelines for deployment. This isn't a static textbook—it's a living, evolving resource designed to keep pace with advancements in the field.

> "If you want to go fast, go alone. If you want to go far, go together."
> – African Proverb

As a living and breathing resource, this book is a continual work in progress, reflecting the ever-evolving nature of machine learning systems. Advancements in the ML landscape drive our commitment to keeping this resource updated with the latest insights, techniques, and best practices. We warmly invite you to join us on this journey by contributing your expertise, feedback, and ideas.

Global Outreach

Thank you to all our readers and visitors. Your engagement with the material keeps us motivated.

Why We Wrote This Book

While there are plenty of resources that focus on the algorithmic side of machine learning, resources on the systems side of things are few and far between. This gap inspired us to create this book—a resource dedicated to the principles and practices of building efficient and scalable ML systems.

Our vision for this book and its broader mission is deeply rooted in the transformative potential of AI and the need to make AI education globally accessible to all. To learn more about the inspiration behind this project and the values driving its creation, we encourage you to read the Author's Note.

Want to Help Out?

This is a collaborative project, and your input matters! If you'd like to contribute, check out our contribution guidelines. Feedback, corrections, and new ideas are welcome. Simply file a GitHub issue.

What's Next?

If you're ready to dive deeper into the book's structure, learning objectives, and practical use, visit the About the Book section for more details.

FRONTMATTER

Author's Note

AI is bound to transform the world in profound ways, much like computers and the Internet revolutionized every aspect of society in the 20th century. From systems that generate creative content to those driving breakthroughs in drug discovery, AI is ushering in a new era—one that promises to be even more transformative in its scope and impact. But how do we make it accessible to everyone?

With its transformative power comes an equally great responsibility for those who access it or work with it. Just as we expect companies to wield their influence ethically, those of us in academia bear a parallel responsibility: to share our knowledge openly, so it benefits everyone—not just a select few. This conviction inspired the creation of this book—an open-source resource aimed at making AI education, particularly in AI engineering, and systems, inclusive, and accessible to everyone from all walks of life.

My passion for creating, curating, and editing this content has been deeply influenced by landmark textbooks that have profoundly shaped both my academic and personal journey. Whether I studied them cover to cover or drew insights from key passages, these resources fundamentally shaped the way I think. I reflect on the books that guided my path: works by Turing Award winners such as David Patterson and John Hennessy—pioneers in computer architecture and system design—and foundational research papers by luminaries like Yann LeCun, Geoffrey Hinton, and Yoshua Bengio. In some small part, my hope is that this book will inspire students to chart their own unique paths.

I am optimistic about what lies ahead for AI. It has the potential to solve global challenges and unlock creativity in ways we have yet to imagine. To achieve this, however, we must train the next generation of AI engineers and practitioners—those who can transform novel AI algorithms into working systems that enable real-world application. This book is a step toward curating the material needed to build the next generation of AI engineers who will transform today's visions into tomorrow's reality.

This book is a work in progress, but knowing that even one learner benefits from its content motivates me to continually refine and expand it. To that end, if there's one thing I ask of readers, it's this: please show your support by starring the GitHub repository here. Your star reflects your belief in this mission—not just to me, but to the growing global community of learners, educators, and practitioners. This small act is more than symbolic—it amplifies the importance of making AI education accessible.

I am a student of my own writing, and every chapter of this book has taught me something new—thanks to the numerous people who have played, and continue to play, an important role in shaping this work. Professors, students, practitioners, and researchers contributed by offering suggestions, sharing expertise, identifying errors, and proposing improvements. Every interaction, whether a detailed critique or a simple correction from a GitHub contributor, has been a lesson in itself. These contributions have not only refined the material but also deepened my understanding of how knowledge grows through collaboration. This book is, therefore, not solely my work; it is a shared endeavor, reflecting the collective spirit of those dedicated to sharing their knowledge and effort.

This book is dedicated to the loving memory of my father. His passion for education, endless curiosity, generosity in sharing knowledge, and unwavering commitment to quality challenge me daily to strive for excellence in all I do. In his honor, I extend this dedication to teachers and mentors everywhere, whose efforts and guidance transform lives every day. Your selfless contributions remind me to persevere.

Last but certainly not least, this work would not be possible without the unwavering support of my wonderful wife and children. Their love, patience, and encouragement form the foundation that enables me to pursue my passion and bring this work to life. For this, and so much more, I am deeply grateful.

— Prof. Vijay Janapa Reddi

About the Book

Overview

Purpose of the Book

Welcome to this collaborative textbook. It originated as part of the *CS249r: Tiny Machine Learning* course that Prof. Vijay Janapa Reddi teaches at Harvard University.

The goal of this book is to provide a resource for educators and learners seeking to understand the principles and practices of machine learning systems. This book is continually updated to incorporate the latest insights and effective teaching strategies with the intent that it remains a valuable resource in this fast-evolving field. So please check back often!

Context and Development

The book originated as a collaborative effort with contributions from students, researchers, and practitioners. While maintaining its academic rigor and real-world applicability, it continues to evolve through regular updates and careful curation to reflect the latest developments in machine learning systems.

What to Expect

This textbook explores the foundational principles, practical workflows, and critical challenges of building and deploying machine learning systems. Starting with **foundational concepts**, it progresses through **engineering principles**, examines **operational considerations** for deploying AI systems, and concludes by reflecting on the societal and technological implications of machine learning.

Learning Goals

Key Learning Outcomes

This book is structured with Bloom's Taxonomy in mind, which defines six levels of learning, ranging from foundational knowledge to advanced creative thinking:

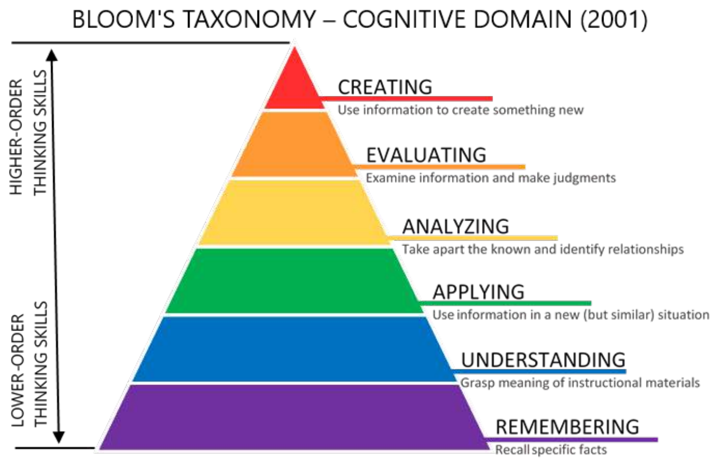

BLOOM'S TAXONOMY – COGNITIVE DOMAIN (2001)
HIGHER-ORDER THINKING SKILLS
LOWER-ORDER THINKING SKILLS
CREATING
Use information to create something new
EVALUATING
Examine information and make judgments
ANALYZING
Take apart the known and identify relationships
APPLYING
Use information in a new (but similar) situation
UNDERSTANDING
Grasp meaning of instructional materials
REMEMBERING
Recall specific facts

Figure 0.1: Bloom's Taxonomy (2021 edition).

1. **Remembering**: Recalling basic facts and concepts.
2. **Understanding**: Explaining ideas or processes.
3. **Applying**: Using knowledge in new situations.
4. **Analyzing**: Breaking down information into components.
5. **Evaluating**: Making judgments based on criteria and standards.
6. **Creating**: Producing original work or solutions.

Learning Objectives

This book supports readers in:

1. **Understanding Fundamentals**: Explain the foundational principles of machine learning, including theoretical underpinnings and practical applications.
2. **Analyzing System Components**: Evaluate the critical components of AI systems and their roles within various architectures.
3. **Designing Workflows**: Outline workflows for developing machine learning systems, from data collection to deployment.
4. **Optimizing Models**: Apply methods to enhance performance, such as hyperparameter tuning and regularization.
5. **Evaluating Ethical Implications**: Analyze societal impacts and address potential biases in AI systems.

6. **Exploring Applications**: Investigate real-world use cases across diverse domains.
7. **Considering Deployment Challenges**: Address security, scalability, and maintainability in real-world systems.
8. **Envisioning Future Trends**: Reflect on emerging challenges and technologies in machine learning.

AI Learning Companion

Throughout this resource, you'll find **SocratiQ**—an AI learning assistant designed to enhance your learning experience. Inspired by the Socratic method of teaching, SocratiQ combines interactive quizzes, personalized assistance, and real-time feedback to help you reinforce your understanding and create new connections. As part of our experiment with Generative AI technologies, SocratiQ encourages critical thinking and active engagement with the material.

SocratiQ is still a work in progress, and we welcome your feedback to make it better. For more details about how SocratiQ works and how to get the most out of it, visit the AI Learning Companion page.

How to Use This Book

Book Structure

The book is organized into four main parts, each building on the previous one:

1. **The Essentials (Chapters 1-4)** Core principles, components, and architectures that underpin machine learning systems.
2. **Engineering Principles (Chapters 5-13)** Covers workflows, data engineering, optimization strategies, and operational challenges in system design.
3. **AI Best Practice (Chapters 14-18)** Focuses on key considerations for deploying AI systems in real-world environments, including security, privacy, robustness, and sustainability.
4. **Closing Perspectives (Chapter 19-20)** Synthesizes key lessons and explores emerging trends shaping the future of ML systems.

Suggested Reading Paths

- **Beginners**: Start with *The Essentials* to build a strong conceptual base before progressing to other parts.
- **Practitioners**: Focus on *Engineering Principles* and *AI in Practice* for hands-on, real-world insights.
- **Researchers**: Dive into *AI in Practice* and *Closing Perspectives* to explore advanced topics and societal implications.

Modular Design

The book is modular, allowing readers to explore chapters independently or sequentially. Each chapter includes supplementary resources:

- **Slides** summarizing key concepts.
- **Videos** providing in-depth explanations.
- **Exercises** reinforcing understanding.
- **Labs** offering practical, hands-on experience.

While several of these resources are still a work in progress, we believe it's better to share valuable insights and tools as they become available rather than wait for everything to be perfect. After all, progress is far more important than perfection, and your feedback will help us improve and refine this resource over time.

Additionally, we try to reuse and build upon the incredible work created by amazing experts in the field, rather than reinventing everything from scratch. This philosophy reflects the fundamental essence of community-driven learning: collaboration, sharing knowledge, and collectively advancing our understanding.

Transparency and Collaboration

This book is a community-driven project, with content generated collaboratively by numerous contributors over time. The content creation process may have involved various editing tools, including generative AI technology. As the main author, editor, and curator, Prof. Vijay Janapa Reddi maintains human oversight to ensure the content is accurate and relevant.

However, no one is perfect, and inaccuracies may still exist. Your feedback is highly valued, and we encourage you to provide corrections or suggestions. This collaborative approach is crucial for maintaining high-quality information and making it globally accessible.

Copyright and Licensing

This book is open-source and developed collaboratively through GitHub. Unless otherwise stated, this work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).

Contributors retain copyright over their individual contributions, dedicated to the public domain or released under the same open license as the original project. For more information on authorship and contributions, visit the GitHub repository.

Join the Community

This textbook is more than just a resource—it's an invitation to collaborate and learn together. Engage in community discussions to share insights, tackle challenges, and learn alongside fellow students, researchers, and practitioners.

Whether you're a student starting your journey, a practitioner solving real-world challenges, or a researcher exploring advanced concepts, your contributions will enrich this learning community. Introduce yourself, share your goals, and let's collectively build a deeper understanding of machine learning systems.

Book Changelog

This Machine Learning Systems textbook is constantly evolving. This changelog automatically records all updates and improvements, helping you stay informed about what's new and refined.

For the complete and most up-to-date changelog, please visit mlsysbook.ai.

Acknowledgements

This book, inspired by the TinyML edX course and CS294r at Harvard University, is the result of years of hard work and collaboration with many students, researchers and practitioners. We are deeply indebted to the folks whose groundbreaking work laid its foundation.

As our understanding of machine learning systems deepened, we realized that fundamental principles apply across scales, from tiny embedded systems to large-scale deployments. This realization shaped the book's expansion into an exploration of machine learning systems with the aim of providing a foundation applicable across the spectrum of implementations.

Funding Agencies and Companies

Academic Support

We are grateful for the academic support that has made it possible to hire teaching assistants to help improve instructional material and quality:

HDSI | Harvard Data Science Initiative

HARVARD Extension School

NSF

Non-Profit and Institutional Support

We gratefully acknowledge the support of the following non-profit organizations and institutions that have contributed to educational outreach efforts, provided scholarship funds to students in developing countries, and organized workshops to teach using the material:

Corporate Support

The following companies contributed hardware kits used for the labs in this book and/or supported the development of hands-on educational materials:

Contributors

We express our sincere gratitude to the open-source community of learners, educators, and contributors. Each contribution, whether a chapter section or a single-word correction, has significantly enhanced the quality of this resource. We also acknowledge those who have shared insights, identified issues, and provided valuable feedback behind the scenes.

A comprehensive list of all GitHub contributors, automatically updated with each new contribution, is available below. For those interested in contributing further, please consult our GitHub page for more information.

Vijay Janapa Reddi
jasonjabbour
Ikechukwu Uchendu
Zeljko Hrcek
Kai Kleinbard
Naeem Khoshnevis
Marcelo Rovai
Sara Khosravi
Douwe den Blanken
shanzehbatool
Elias
Jared Ping
Jeffrey Ma
Itai Shapira
Maximilian Lam
Jayson Lin
Sophia Cho
Andrea

Alex Rodriguez
Korneel Van den Berghe
Zishen Wan
Colby Banbury
Mark Mazumder
Divya Amirtharaj
Abdulrahman Mahmoud
Srivatsan Krishnan
marin-llobet
Emeka Ezike
Aghyad Deeb
Haoran Qiu
Aditi Raju
ELSuitorHarvard
Emil Njor
Michael Schnebly
Jared Ni
oishib
Yu-Shun Hsiao
Jae-Won Chung
Henry Bae
Jennifer Zhou
Arya Tschand
Eura Nofshin
Pong Trairatvorakul
Matthew Stewart
Marco Zennaro
Andrew Bass
Shvetank Prakash
Fin Amin
Allen-Kuang
Gauri Jain
gnodipac886
The Random DIY
Bruno Scaglione
Fatima Shah
Sercan Aygün
Alex Oesterling
Baldassarre Cesarano
Abenezer
TheHiddenLayer
abigailswallow
yanjingl
happyappledog
Yang Zhou
Aritra Ghosh
Andy Cheng
Bilge Acun

Jessica Quaye
Jason Yik
Emmanuel Rassou
Shreya Johri
Sonia Murthy
Vijay Edupuganti
Costin-Andrei Oncescu
Annie Laurie Cook
Jothi Ramaswamy
Batur Arslan
Curren Iyer
Fatima Shah
Edward Jin
a-saraf
songhan
Zishen

SocratiQ AI

AI Learning Companion

Welcome to SocratiQ (pronounced "Socratic' '), an AI learning assistant seamlessly integrated throughout this resource. Inspired by the Socratic method of teaching—emphasizing thoughtful questions and answers to stimulate critical thinking—SocratiQ is part of our experiment with what we call as *Generative Learning*. By combining interactive quizzes, personalized assistance, and real-time feedback, SocratiQ is meant to reinforce your understanding and help you create new connections. *SocratiQ is still a work in progress, and we welcome your feedback.*

Learn more: Read our research paper on SocratiQ's design and pedagogy here.

Listen to this AI-generated podcast about SocratiQ here.

You can enable SocratiQ by clicking the button below:

SocratiQ: OFF

> 💡 Direct URL Access
>
> You can directly control SocratiQ by adding `?socratiq=` parameters to your URL:
>
> - To activate: mlsysbook.ai/?socratiq=true
> - To deactivate: mlsysbook.ai/?socratiq=false
>
> This gives you with quick access to toggle SocratiQ's functionality directly from your browser's address bar if you are on a page and do not want to return here to toggle functionality.

SocratiQ's goal is to adapt to your needs while generating targeted questions and engaging in meaningful dialogue about the material. Unlike traditional textbook study, SocratiQ offers an interactive, personalized learning experience that can help you better understand and retain complex concepts. It is only available as an online feature.

Quick Start Guide

1. Enable SocratiQ using the button below or URL parameters

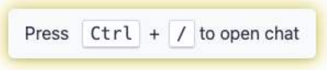
2. Use keyboard shortcut (`Cmd/Ctrl + /`) to open SocratiQ anytime
3. Set your academic level in Settings
4. Start learning! Look for quiz buttons at the end of sections

Please note that this is an experimental feature. We are experimenting with the idea of creating a dynamic and personalized learning experience by harnessing the power of generative AI. We hope that this approach will transform how you interact with and absorb the complex concepts.

> ⚠ Warning
>
> **About AI Responses**: While SocratiQ uses advanced AI to generate quizzes and provide assistance, like all AI systems, it may occasionally provide imperfect or incomplete answers. However, we've designed and tested it to ensure it's effective for supporting your learning journey. If you're unsure about any response, refer to the textbook content or consult your instructor.

Once you've enabled SocratiQ it will always be available when you visit this site.

You can access SocratiQ at any time using a keyboard shortcut shown in Figure 0.2, which brings up the interface shown in Figure 0.3.



Press Ctrl + / to open chat

Figure 0.2: Keyboard shortcut for SocratiQ.

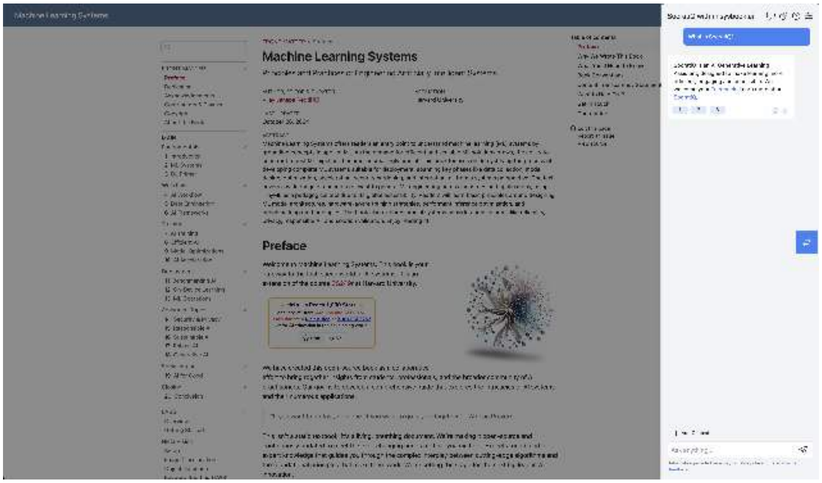


Figure 0.3: The main SocratiQ interface, showing the key components of your AI learning assistant.

Button Overview

The top nav bar provides quick access to the following features:

1. Adjust your settings at any time.
2. Track your progress by viewing the dashboard.
3. Start new or save your conversations with SocratiQ.

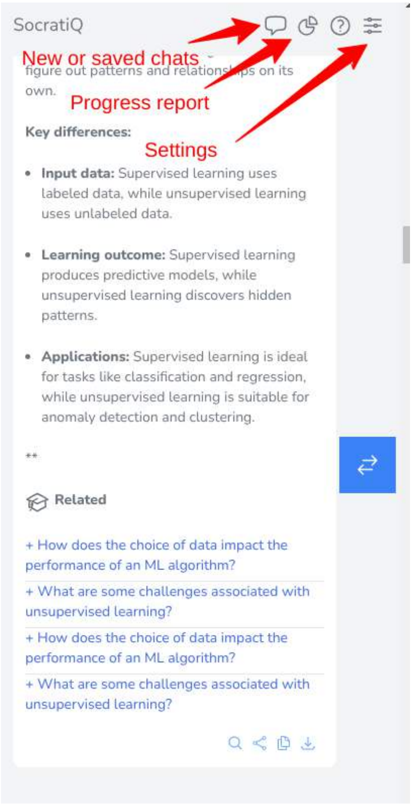


figure out patterns and relationships on its own.

Key differences:

- **Input data:** Supervised learning uses labeled data, while unsupervised learning uses unlabeled data.
- **Learning outcome:** Supervised learning produces predictive models, while unsupervised learning discovers hidden patterns.
- **Applications:** Supervised learning is ideal for tasks like classification and regression, while unsupervised learning is suitable for anomaly detection and clustering.

**

+ How does the choice of data impact the performance of an ML algorithm?
+ What are some challenges associated with unsupervised learning?
+ How does the choice of data impact the performance of an ML algorithm?
+ What are some challenges associated with unsupervised learning?

Figure 0.4: View of the top nav menu.

Personalize Your Learning

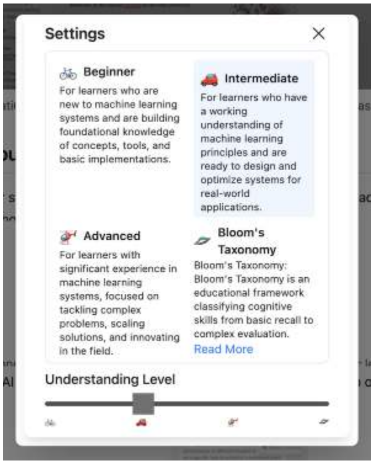
Before diving into your studies, take a moment to configure SocratiQ for your academic level. This initial setup ensures that all interactions, from quiz questions to explanations, are tailored to your background knowledge. Figure 0.5 shows where you can adjust these preferences.

You can augment any AI SocratiQ response using the dropdown menu at the top of each message.

Learning with SocratiQ

Quizzes

As you progress through each section of the textbook, you have the option to ask SocratiQ to automatically generate quizzes tailored to reinforce key concepts.



Settings

Beginner
For learners who are new to machine learning systems and are building foundational knowledge of concepts, tools, and basic implementations.

Intermediate
For learners who have a working understanding of machine learning principles and are ready to design and optimize systems for real-world applications.

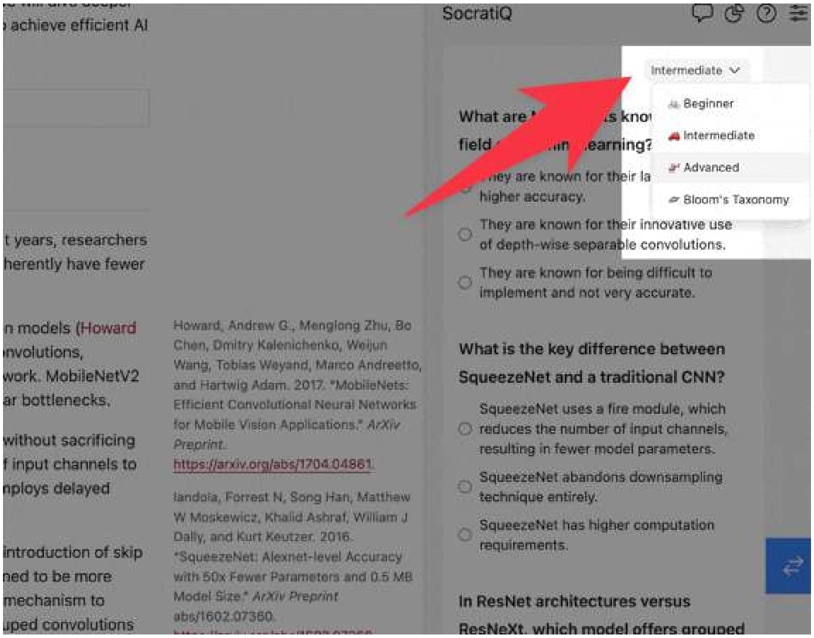
Advanced
For learners with significant experience in machine learning systems, focused on tackling complex problems, scaling solutions, and innovating in the field.

Bloom's Taxonomy
Bloom's Taxonomy: Bloom's Taxonomy is an educational framework classifying cognitive skills from basic recall to complex evaluation.
Read More

Understanding Level

Figure 0.5: The settings panel where you can customize SocratiQ to match your academic level.

These quizzes are conveniently inserted at the end of every major subsection (e.g., 1.1, 1.2, 1.3, and so on), as illustrated in Figure 0.7.



achieve efficient AI

t years, researchers
herently have fewer

n models (Howard
nvolutions,
work. MobileNetV2
ar bottlenecks.

without sacrificing
f input channels to
nploys delayed

introduction of skip
ned to be more
mechanism to
uped convolutions

Howard, Andrew G., Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *ArXiv Preprint*. https://arxiv.org/abs/1704.04861.

Iandola, Forrest N, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. "SqueezeNet: Alexnet-level Accuracy with 50x Fewer Parameters and 0.5 MB Model Size." *ArXiv Preprint* abs/1602.07360.

SocratiQ

Intermediate

- Beginner
- Intermediate
- Advanced
- Bloom's Taxonomy

What are ... s kno field ... in earning?

○ ...ey are known for their la higher accuracy.

○ They are known for their innovative use of depth-wise separable convolutions.

○ They are known for being difficult to implement and not very accurate.

What is the key difference between SqueezeNet and a traditional CNN?

○ SqueezeNet uses a fire module, which reduces the number of input channels, resulting in fewer model parameters.

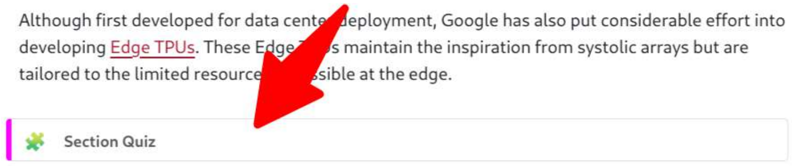
○ SqueezeNet abandons downsampling technique entirely.

○ SqueezeNet has higher computation requirements.

In ResNet architectures versus ResNeXt, which model offers grouped

Figure 0.6: Redo an AI message by choosing a new experience level.

Each quiz typically consists of 3-5 multiple-choice questions and takes only 1-2 minutes to complete. These questions are designed to assess your understanding of the material covered in the preceding section, as shown in Figure 0.8a.

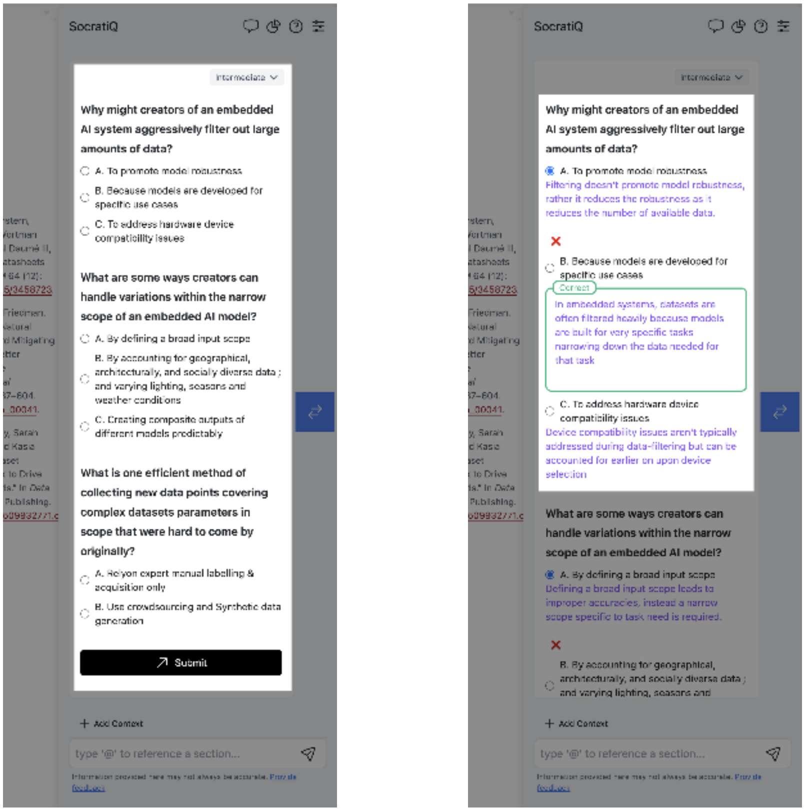


Although first developed for data cente deployment, Google has also put considerable effort into developing Edge TPUs. These Edge T s maintain the inspiration from systolic arrays but are tailored to the limited resource sible at the edge.

Section Quiz

Figure 0.7: Quizzes are generated at the end of every section.

Upon submitting your answers, SocratiQ provides immediate feedback along with detailed explanations for each question, as demonstrated in Figure 0.8b.



SocratiQ
Intermediate
Submit
Add Context
type '@' to reference a section...
Information provided here may not always be accurate. Provide feedback

Why might creators of an embedded AI system aggressively filter out large amounts of data?

○ A. To promote model robustness

○ B. Because models are developed for specific use cases

○ C. To address hardware device compatibility issues

What are some ways creators can handle variations within the narrow scope of an embedded AI model?

○ A. By defining a broad input scope

○ B. By accounting for geographical, architecturally, and socially diverse data ; and varying lighting, seasons and weather conditions

○ C. Creating composite outputs of different models predictably

What is one efficient method of collecting new data points covering complex datasets parameters in scope that were hard to come by originally?

○ A. Relyon expert manual labelling & acquisition only

○ B. Use crowdsourcing and Synthetic data generation

(a) Example of AI-generated quiz questions.

Why might creators of an embedded AI system aggressively filter out large amounts of data?

● A. To promote model robustness

Filtering doesn't promote model robustness, rather it reduces the robustness as it reduces the number of available data.

✗

○ B. Because models are developed for specific use cases

Correct

In embedded systems, datasets are often filtered heavily because models are built for very specific tasks narrowing down the data needed for that task

○ C. To address hardware device compatibility issues

Device compatibility issues aren't typically addressed during data-filtering but can be accounted for earlier on upon device selection

What are some ways creators can handle variations within the narrow scope of an embedded AI model?

● A. By defining a broad input scope

Defining a broad input scope leads to improper accuracies, instead a narrow scope specific to task need is required.

✗

○ B. By accounting for geographical, architecturally, and socially diverse data ; and varying lighting, seasons and

(b) Example of AI-generated feedback and explanations for quizzes.

Figure 0.8: SocratiQ uses a Large Language Model (LLM) to automatically generate and grade quizzes.

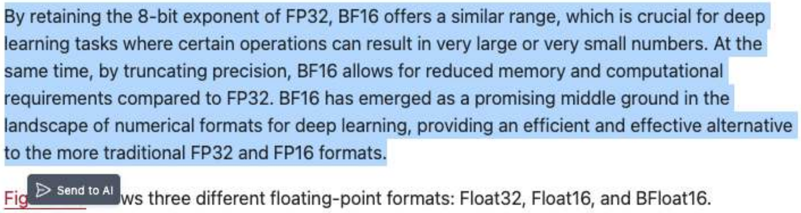
Example Learning Flow

1. Read a section
2. Select challenging text → Ask SocratiQ for explanation
3. Take the section quiz
4. Review related content suggestions

5. Track progress in dashboard

Getting Help with Concepts

When you encounter challenging concepts, SocratiQ offers two powerful ways to get help. First, you can select any text from the textbook and ask for a detailed explanation, as demonstrated in Figure 0.9.

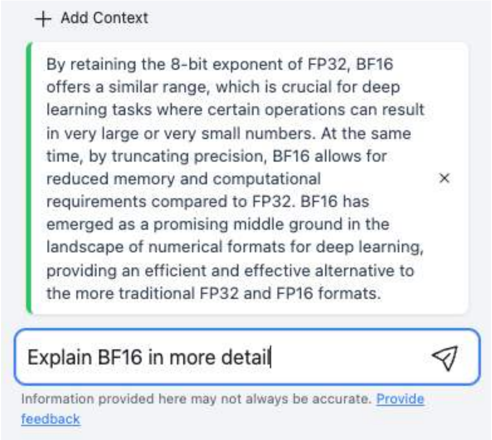


By retaining the 8-bit exponent of FP32, BF16 offers a similar range, which is crucial for deep learning tasks where certain operations can result in very large or very small numbers. At the same time, by truncating precision, BF16 allows for reduced memory and computational requirements compared to FP32. BF16 has emerged as a promising middle ground in the landscape of numerical formats for deep learning, providing an efficient and effective alternative to the more traditional FP32 and FP16 formats.

Fig ws three different floating-point formats: Float32, Float16, and BFloat16.

Figure 0.9: Selecting specific text to ask for clarification.

Once you've selected the text, you can ask questions about it, and SocratiQ will provide detailed explanations based on that context, as illustrated in Figure 0.10.



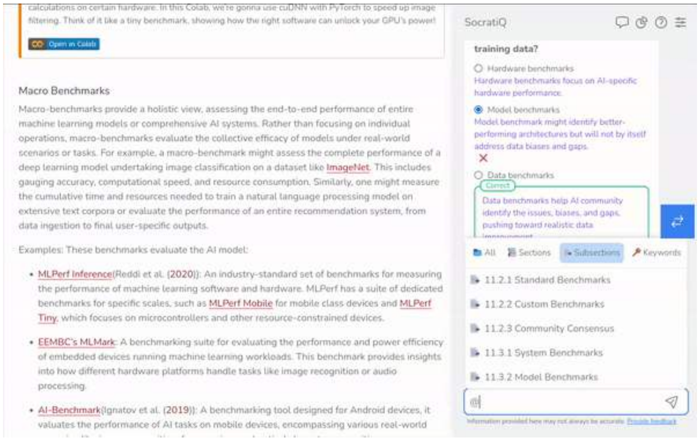
By retaining the 8-bit exponent of FP32, BF16 offers a similar range, which is crucial for deep learning tasks where certain operations can result in very large or very small numbers. At the same time, by truncating precision, BF16 allows for reduced memory and computational requirements compared to FP32. BF16 has emerged as a promising middle ground in the landscape of numerical formats for deep learning, providing an efficient and effective alternative to the more traditional FP32 and FP16 formats.

Explain BF16 in more detail

Information provided here may not always be accurate. Provide feedback

Figure 0.10: Example of how SocratiQ provides explanations based on selected text.

Figure 0.12 shows the response for the ask in Figure 0.10.



calculations on certain hardware. In this Colab, we're gonna use cuDNN with PyTorch to speed up image filtering. Think of it like a tiny benchmark, showing how the right software can unlock your GPU's power!

Macro Benchmarks

Macro-benchmarks provide a holistic view, assessing the end-to-end performance of entire machine learning models or comprehensive AI systems. Rather than focusing on individual operations, macro-benchmarks evaluate the collective efficacy of models under real-world scenarios or tasks. For example, a macro-benchmark might assess the complete performance of a deep learning model undertaking image classification on a dataset like ImageNet. This includes gauging accuracy, computational speed, and resource consumption. Similarly, one might measure the cumulative time and resources needed to train a natural language processing model on extensive text corpora or evaluate the performance of an entire recommendation system, from data ingestion to final user-specific outputs.

Examples: These benchmarks evaluate the AI model:

- MLPerf Inference(Reddi et al. (2020)): An industry-standard set of benchmarks for measuring the performance of machine learning software and hardware. MLPerf has a suite of dedicated benchmarks for specific scales, such as MLPerf Mobile for mobile class devices and MLPerf Tiny, which focuses on microcontrollers and other resource-constrained devices.
- EEMBC's MLMark: A benchmarking suite for evaluating the performance and power efficiency of embedded devices running machine learning workloads. This benchmark provides insights into how different hardware platforms handle tasks like image recognition or audio processing.
- AI-Benchmark(Ignatov et al. (2019)): A benchmarking tool designed for Android devices, it valuates the performance of AI tasks on mobile devices, encompassing various real-world

training data?

○ Hardware benchmarks

Hardware benchmarks focus on AI-specific hardware performance.

● Model benchmarks

Model benchmark might identify better-performing architectures but will not by itself address data biases and gaps.

✗

○ Data benchmarks

Data benchmarks help AI community identify the issues, biases, and gaps, pushing toward realistic data improvement

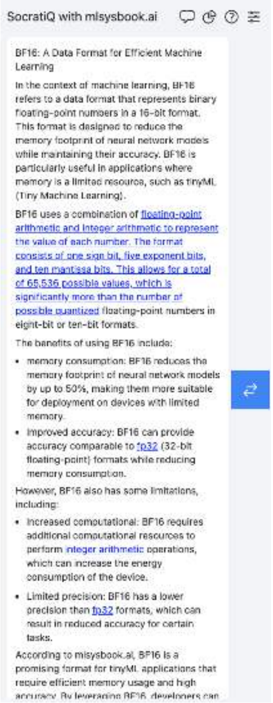
- 11.2.1 Standard Benchmarks
- 11.2.2 Custom Benchmarks
- 11.2.3 Community Consensus
- 11.3.1 System Benchmarks
- 11.3.2 Model Benchmarks

@|

Information provided here may not always be accurate. Provide feedback

Figure 0.11: Referencing different sections from the textbook.

Additionally, you can also reference Sections, as shown in Figure 0.11, Subsections and keywords directly as you converse with SocratiQ. Use the `@` symbol to reference a section, sub-section or keyword. You can also click the `+ Context` button right above the input.



BF16: A Data Format for Efficient Machine Learning

In the context of machine learning, BF16 refers to a data format that represents binary floating-point numbers in a 16-bit format. This format is designed to reduce the memory footprint of neural network models while maintaining their accuracy. BF16 is particularly useful in applications where memory is a limited resource, such as tinyML (Tiny Machine Learning).

BF16 uses a combination of floating-point arithmetic and integer arithmetic to represent the value of each number. The format consists of one sign bit, five exponent bits, and ten mantissa bits. This allows for a total of 65,536 possible values, which is significantly more than the number of possible quantized floating-point numbers in eight-bit or ten-bit formats.

The benefits of using BF16 include:

- memory consumption: BF16 reduces the memory footprint of neural network models by up to 50%, making them more suitable for deployment on devices with limited memory.
- Improved accuracy: BF16 can provide accuracy comparable to fp32 (32-bit floating-point) formats while reducing memory consumption.

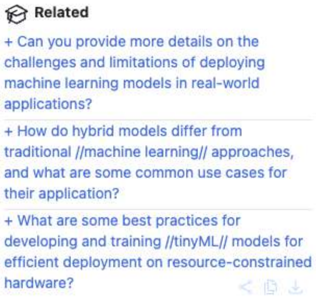
However, BF16 also has some limitations, including:

- Increased computational: BF16 requires additional computational resources to perform integer arithmetic operations, which can increase the energy consumption of the device.
- Limited precision: BF16 has a lower precision than fp32 formats, which can result in reduced accuracy for certain tasks.

According to mlsysbook.ai, BF16 is a promising format for tinyML applications that require efficient memory usage and high accuracy. By leveraging BF16, developers can

Figure 0.12: An interactive chat session with SocratiQ, demonstrating how to get clarification on concepts.

To enhance your learning experience, SocratiQ doesn't just answer your questions, it also suggests related content from the textbook that might be helpful for deeper understanding, as shown in Figure 0.13.



Related

+ Can you provide more details on the challenges and limitations of deploying machine learning models in real-world applications?

+ How do hybrid models differ from traditional //machine learning// approaches, and what are some common use cases for their application?

+ What are some best practices for developing and training //tinyML// models for efficient deployment on resource-constrained hardware?

Figure 0.13: SocratiQ suggests related content based on your questions to help deepen your understanding.

Tracking Your Progress

Performance Dashboard

SocratiQ maintains a comprehensive record of your learning journey. The progress dashboard (Figure 0.14) displays your quiz performance statistics, learning streaks, and achievement badges. This dashboard updates real-time.

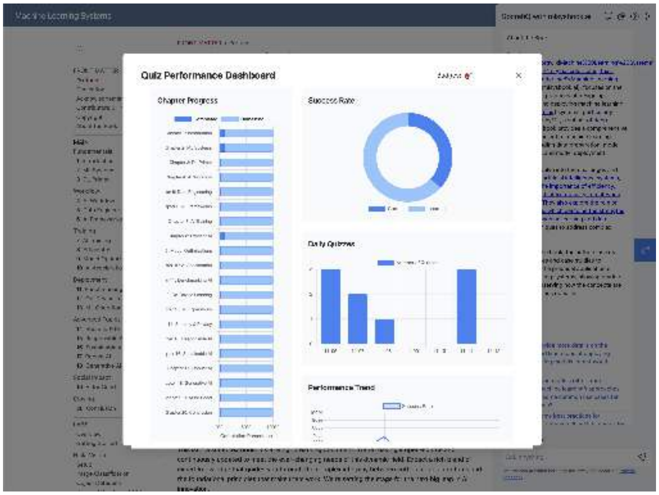


Figure 0.14: The progress dashboard showing your learning statistics and achievements.

As you continue to engage with the material and complete quizzes, you'll earn various badges that recognize your progress, as shown in Figure 0.15.

Badges: 🎯

Figure 0.15: Examples of achievement badges you can earn through consistent engagement.

🏅 Achievement Badges

As you progress through the quizzes, you'll earn special badges to mark your achievements! Here's what you can earn:

Badge	Name	How to Earn
🎯	First Steps	Complete your first quiz
🔢	On a Streak	Maintain a streak of perfect scores
🏆	Quiz Medalist	Complete 10 quizzes
🏆🏆	Quiz Champion	Complete 20 quizzes
🏆🏆🏆	Quiz Legend	Complete 30 quizzes
🏆🏆🏆🏆 x n	Quiz AGI Super Human	Complete 40 or more quizzes

💡 Tip

Keep taking quizzes to collect all badges and improve your learning journey! Your current badges will appear in the quiz statistics dashboard.

If you'd like a record of your progress you can generate a PDF report. It will show your progress, average performance and all the questions you've attempted. The PDF is a generated with a unique hash and can be uniquely validated.

Data Storage

❗ Important

Important Note: All progress data is stored locally in your browser. Clearing your browser history or cache will erase your entire learning history, including quiz scores, streaks, and achievement badges.

You can also delete all of your saved conversations by clicking the New Chat button in the nav bar.

Technical Requirements

To use SocratiQ effectively, you'll need:

- Chrome or Safari browser
- JavaScript enabled
- Stable internet connection

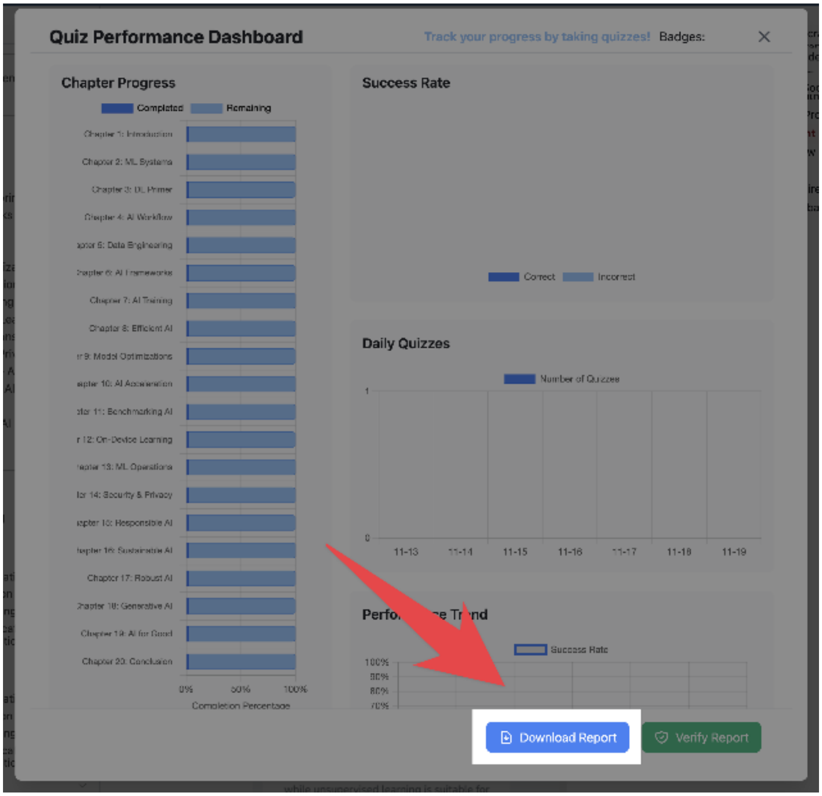


Figure 0.16: You can click the Download Report button to view your report. You can verify that your PDF has been created by SocratiQ by clicking the verify button and uploading your generated PDF.

Common Issues and Troubleshooting

- If SocratiQ isn't responding: Refresh the page
- If quizzes don't load: Check your internet connection
- If progress isn't saving: Ensure cookies are enabled

For persistent issues, please contact us at vj[@]eecs.harvard.edu.

Providing Feedback

Your feedback helps us improve SocratiQ.

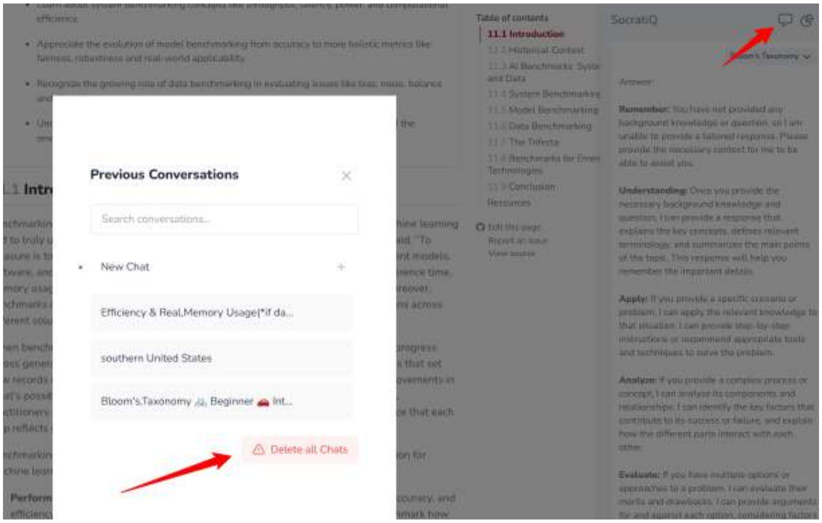


Figure 0.17: Load or delete previous chats or start a new chat.

You can report technical issues, suggest improvements to quiz questions, or share thoughts about AI responses using the feedback buttons located throughout the interface. You can submit a GitHub issue.

MAIN

Chapter 1

Introduction

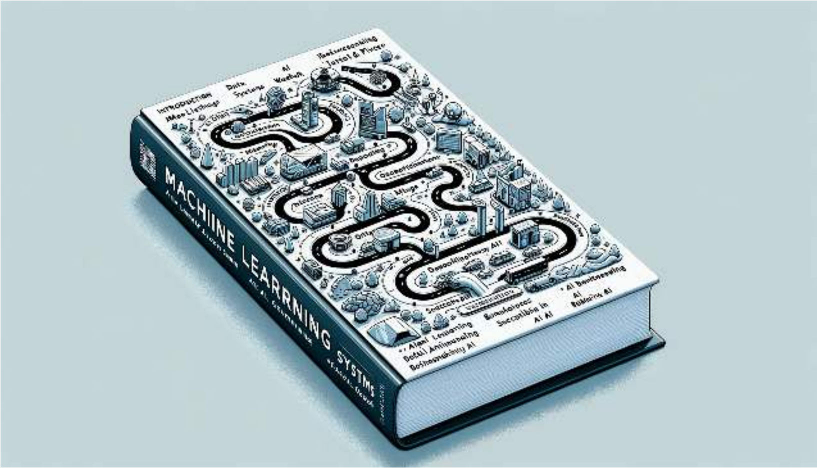


Figure 1.1: *DALL·E 3 Prompt: A detailed, rectangular, flat 2D illustration depicting a roadmap of a book's chapters on machine learning systems, set on a crisp, clean white background. The image features a winding road traveling through various symbolic landmarks. Each landmark represents a chapter topic: Introduction, ML Systems, Deep Learning, AI Workflow, Data Engineering, AI Frameworks, AI Training, Efficient AI, Model Optimizations, AI Acceleration, Benchmarking AI, On-Device Learning, Embedded AIOps, Security & Privacy, Responsible AI, Sustainable AI, AI for Good, Robust AI, Generative AI. The style is clean, modern, and flat, suitable for a technical book, with each landmark clearly labeled with its chapter title.*

1.1 AI Pervasiveness

Artificial Intelligence (AI) has emerged as one of the most transformative forces in human history. From the moment we wake up to when we go to sleep, AI systems invisibly shape our world. They manage traffic flows in our cities, optimize power distribution across electrical grids, and enable billions of wireless devices to communicate seamlessly. In hospitals, AI analyzes medical images and helps doctors diagnose diseases. In research laboratories, it accelerates scientific discovery by simulating molecular interactions and processing vast datasets from particle accelerators. In space exploration, it helps rovers navigate distant planets and telescopes detect new celestial phenomena.

Throughout history, certain technologies have fundamentally transformed human civilization, defining their eras. The 18th and 19th centuries were shaped by the Industrial Revolution, where steam power and mechanization transformed how humans could harness physical energy. The 20th century was defined by the Digital Revolution, where the computer and internet transformed how we process and share information. Now, the 21st century appears to be the era of Artificial Intelligence, a shift noted by leading thinkers in technological evolution (Brynjolfsson and McAfee 2014; Domingos 2016).

The vision driving AI development extends far beyond the practical applications we see today. We aspire to create systems that can work alongside humanity, enhancing our problem-solving capabilities and accelerating scientific progress. Imagine AI systems that could help us understand consciousness, decode the complexities of biological systems, or unravel the mysteries of dark matter. Consider the potential of AI to help address global challenges like climate change, disease, or sustainable energy production. This is not just about automation or efficiency—it's about expanding the boundaries of human knowledge and capability.

The impact of this revolution operates at multiple scales, each with profound implications. At the individual level, AI personalizes our experiences and augments our daily decision-making capabilities. At the organizational level, it transforms how businesses operate and how research institutions make discoveries. At the societal level, it reshapes everything from transportation systems to healthcare delivery. At the global level, it offers new approaches to addressing humanity's greatest challenges, from climate change to drug discovery.

What makes this transformation unique is its unprecedented pace. While the Industrial Revolution unfolded over centuries and the Digital Revolution over decades, AI capabilities are advancing at an extraordinary rate. Technologies that seemed impossible just years ago, including systems that can understand human speech, generate novel ideas, or make complex decisions, are now commonplace. This acceleration suggests we are only beginning to understand how profoundly AI will reshape our world.

We stand at a historic inflection point. Just as the Industrial Revolution required us to master mechanical engineering to harness the power of steam and machinery, and the Digital Revolution demanded expertise in electrical and computer engineering to build the internet age, the AI Revolution presents us with a new engineering challenge. We must learn to build systems that can learn, reason, and potentially achieve superhuman capabilities in specific domains.

1.2 AI and ML Basics

The exploration of artificial intelligence's transformative impact across society presents a fundamental question: How can we create these intelligent capabilities? Understanding the relationship between AI and ML provides the theoretical and practical framework necessary to address this question.

Artificial Intelligence represents the systematic pursuit of understanding and replicating intelligent behavior—specifically, the capacity to learn, reason, and adapt to new situations. It encompasses fundamental questions about the nature of intelligence, knowledge, and learning. How do we recognize patterns? How do we learn from experience? How do we adapt our behavior based on new information? AI as a field explores these questions, drawing insights from cognitive science, psychology, neuroscience, and computer science.

Machine Learning, in contrast, constitutes the methodological approach to creating systems that demonstrate intelligent behavior. Instead of implementing intelligence through predetermined rules, machine learning systems utilize gradient descent[0] and other optimization techniques to identify patterns and relationships. This methodology reflects fundamental learning processes observed in biological systems. For instance, object recognition in machine learning systems parallels human visual learning processes, requiring exposure to numerous examples to develop robust recognition capabilities. Similarly, natural language processing systems acquire linguistic capabilities through extensive analysis of textual data.

AI and ML: Key Definitions

- **Artificial Intelligence (AI)**: The goal of creating machines that can match or exceed human intelligence—representing humanity's quest to build systems that can think, reason, and adapt.
- **Machine Learning (ML)**: The scientific discipline of understanding how systems can learn and improve from experience—providing the theoretical foundation for building intelligent systems.

The relationship between AI and ML exemplifies the connection between theoretical understanding and practical engineering implementation observed in other scientific fields. For instance, physics provides the theoretical foundation for mechanical engineering's practical applications in structural design and machinery, while AI's theoretical frameworks inform machine learning's practical development of intelligent systems. Similarly, electrical engineering's transformation of electromagnetic theory into functional power systems parallels machine learning's implementation of intelligence theories into operational ML systems.

The emergence of machine learning as a viable scientific discipline approach to artificial intelligence resulted from extensive research and fundamental paradigm shifts in the field. The progression of artificial intelligence encompasses both theoretical advances in understanding intelligence and practical developments in implementation methodologies. This development mirrors the evolution of other scientific and engineering disciplines—from mechanical engineering's advancement from basic force principles to contemporary robotics, to electrical engineering's progression from fundamental electromagnetic theory to modern power and communication networks. Analysis of this historical trajectory reveals both the technological innovations leading to current machine learning approaches and the emergence of deep reinforcement learning[1] that inform contemporary AI system development.

[0] Gradient Descent: An optimization algorithm that iteratively adjusts model parameters to minimize prediction errors by following the gradient (slope) of the error surface, similar to finding the bottom of a valley by always walking downhill.

[1] Deep Reinforcement Learning: A machine learning approach that combines deep neural networks with reinforcement learning principles, allowing agents to learn optimal actions through trial and error interaction with an environment while receiving rewards or penalties.

1.3 AI Evolution

The evolution of AI, depicted in the timeline shown in Figure 1.2, highlights key milestones such as the development of the perceptron[2] in 1957 by Frank Rosenblatt, a foundational element for modern neural networks. Imagine walking into a computer lab in 1965. You'd find room-sized mainframes running programs that could prove basic mathematical theorems or play simple games like tic-tac-toe. These early artificial intelligence systems, while groundbreaking for their time, were a far cry from today's machine learning systems that can detect cancer in medical images or understand human speech. The timeline shows the progression from early innovations like the ELIZA chatbot in 1966, to significant breakthroughs such as IBM's Deep Blue defeating chess champion Garry Kasparov in 1997. More recent advancements include the introduction of OpenAI's GPT-3 in 2020 and GPT-4 in 2023, demonstrating the dramatic evolution and increasing complexity of AI systems over the decades.

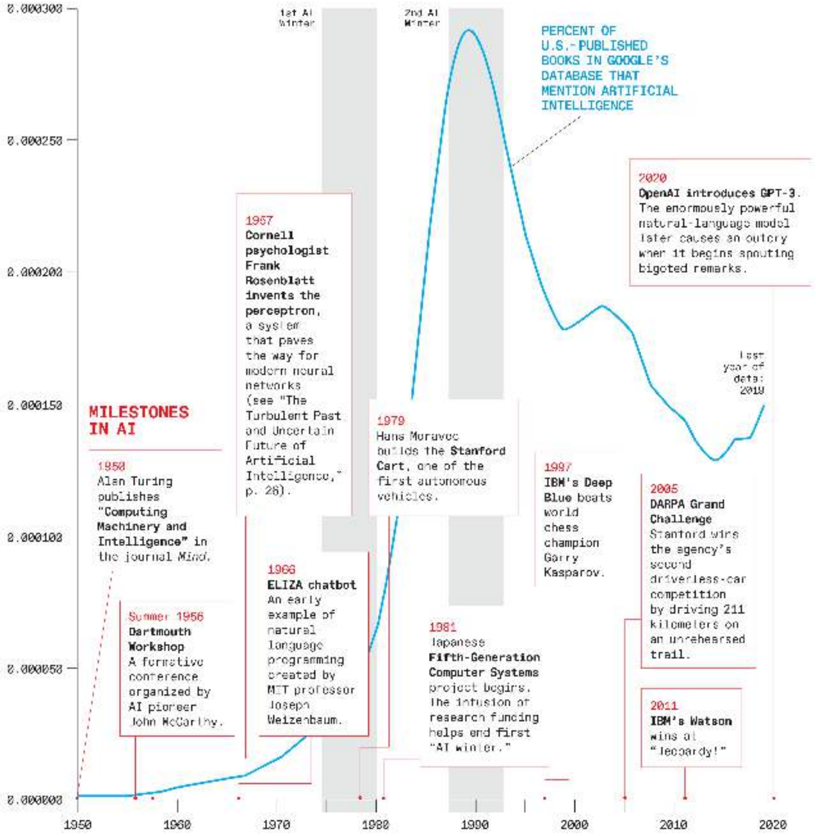


Figure 1.2: Milestones in AI from 1950 to 2020. Source: IEEE Spectrum

Let's explore how we got here.

[2] Perceptron: The first artificial neural network—a simple model that could learn to classify visual patterns, similar to a single neuron making a yes/no decision based on its inputs.

1.3.1 Symbolic AI Era

The story of machine learning begins at the historic Dartmouth Conference in 1956, where pioneers like John McCarthy, Marvin Minsky, and Claude Shannon first coined the term "artificial intelligence." Their approach was based on a compelling idea: intelligence could be reduced to symbol manipulation. Consider Daniel Bobrow's STUDENT system from 1964, one of the first AI programs that could solve algebra word problems. It was one of the first AI programs to demonstrate natural language understanding by converting English text into algebraic equations, marking an important milestone in symbolic AI.

ℹ Example: STUDENT (1964)

```
Problem: "If the number of customers Tom gets is twice the
square of 20% of the number of advertisements he runs, and
the number of advertisements is 45, what is the number of
customers Tom gets?"

STUDENT would:

1. Parse the English text
2. Convert it to algebraic equations
3. Solve the equation: n = 2(0.2 × 45)²
4. Provide the answer: 162 customers
```

Early AI like STUDENT suffered from a fundamental limitation: they could only handle inputs that exactly matched their pre-programmed patterns and rules. Imagine a language translator that only works when sentences follow perfect grammatical structure; even slight variations, such as changing word order, using synonyms, or natural speech patterns, would cause the STUDENT to fail. This "brittleness" meant that while these solutions could appear intelligent when handling very specific cases they were designed for, they would break down completely when faced with even minor variations or real-world complexity. This limitation wasn't just a technical inconvenience—it revealed a deeper problem with rule-based approaches to AI: they couldn't genuinely understand or generalize from their programming, they could only match and manipulate patterns exactly as specified.

1.3.2 Expert Systems Era

By the mid-1970s, researchers realized that general AI was too ambitious. Instead, they focused on capturing human expert knowledge in specific domains. MYCIN, developed at Stanford, was one of the first large-scale expert systems designed to diagnose blood infections.

Example: MYCIN (1976)

```
Rule Example from MYCIN:
IF
  The infection is primary-bacteremia
  The site of the culture is one of the sterile sites
  The suspected portal of entry is the gastrointestinal tract
THEN
  Found suggestive evidence (0.7) that infection is bacteroid
```

While MYCIN represented a major advance in medical AI with its 600 expert rules for diagnosing blood infections, it revealed fundamental challenges that still plague ML today. Getting domain knowledge from human experts and converting it into precise rules proved incredibly time-consuming and difficult—doctors often couldn't explain exactly how they made decisions. MYCIN struggled with uncertain or incomplete information, unlike human doctors who could make educated guesses. Perhaps most importantly, maintaining and updating the rule base became exponentially more complex as MYCIN grew, as adding new rules frequently conflicted with existing ones, while medical knowledge itself continued to evolve. These same challenges of knowledge capture, uncertainty handling, and maintenance remain central concerns in modern machine learning, even though we now use different technical approaches to address them.

1.3.3 Statistical Learning Era

The 1990s marked a radical transformation in artificial intelligence as the field moved away from hand-coded rules toward statistical learning approaches. This wasn't a simple choice—it was driven by three converging factors that made statistical methods both possible and powerful. The digital revolution meant massive amounts of data were suddenly available to train the algorithms. Moore's Law[3] delivered the computational power needed to process this data effectively. And researchers developed new algorithms like Support Vector Machines and improved neural networks that could actually learn patterns from this data rather than following pre-programmed rules. This combination fundamentally changed how we built AI: instead of trying to encode human knowledge directly, we could now let machines discover patterns automatically from examples, leading to more robust and adaptable AI.

Consider how email spam filtering evolved:

Example: Early Spam Detection Systems

```
Rule-based (1980s):
IF contains("viagra") OR contains("winner") THEN spam

Statistical (1990s):
P(spam|word) = (frequency in spam emails) / (total frequency)
```

[3] Moore's Law: The observation made by Intel co-founder Gordon Moore in 1965 that the number of transistors on a microchip doubles approximately every two years, while the cost halves. This exponential growth in computing power has been a key driver of advances in machine learning, though the pace has begun to slow in recent years.

```
Combined using Naive Bayes:
P(spam|email)   P(spam) ×   P(word|spam)
```

The move to statistical approaches fundamentally changed how we think about building AI by introducing three core concepts that remain important today. First, the quality and quantity of training data became as important as the algorithms themselves. AI could only learn patterns that were present in its training examples. Second, we needed rigorous ways to evaluate how well AI actually performed, leading to metrics that could measure success and compare different approaches. Third, we discovered an inherent tension between precision (being right when we make a prediction) and recall (catching all the cases we should find), forcing designers to make explicit trade-offs based on their application's needs. For example, a spam filter might tolerate some spam to avoid blocking important emails, while medical diagnosis might need to catch every potential case even if it means more false alarms.

Table 1.1 encapsulates the evolutionary journey of AI approaches we have discussed so far, highlighting the key strengths and capabilities that emerged with each new paradigm. As we move from left to right across the table, we can observe several important trends. We will talk about shallow and deep learning next, but it is useful to understand the trade-offs between the approaches we have covered so far.

Table 1.1: Evolution of AI, Key Positive Aspects

Aspect	Symbolic AI	Expert Systems	Statistical Learning	Shallow / Deep Learning
Key Strength	Logical reasoning	Domain expertise	Versatility	Pattern recognition
Best Use Case	Well-defined, rule-based problems	Specific domain problems	Various structured data problems	Complex, unstructured data problems
Data Handling	Minimal data needed	Domain knowledge-based	Moderate data required	Large-scale data processing
Adaptability	Fixed rules	Domain-specific adaptability	Adaptable to various domains	Highly adaptable to diverse tasks
Problem Complexity	Simple, logic-based	Complicated, domain- specific	Complex, structured	Highly complex, unstructured

The table serves as a bridge between the early approaches we've discussed and the more recent developments in shallow and deep learning that we'll explore next. It sets the stage for understanding why certain approaches gained prominence in different eras and how each new paradigm built upon and addressed the limitations of its predecessors. Moreover, it illustrates how the strengths of earlier approaches continue to influence and enhance modern AI techniques, particularly in the era of foundation models.

1.3.4 Shallow Learning Era

The 2000s marked a fascinating period in machine learning history that we now call the "shallow learning' ' era. To understand why it's "shallow," imagine building a house: deep learning (which came later) is like having multiple construction crews working at different levels simultaneously, each crew learning from the work of crews below them. In contrast, shallow learning typically had just one or two levels of processing, similar to having just a foundation crew and a framing crew.

During this time, several powerful algorithms dominated the machine learning landscape. Each brought unique strengths to different problems: Decision trees provided interpretable results by making choices much like a flowchart. K-nearest neighbors made predictions by finding similar examples in past data, like asking your most experienced neighbors for advice. Linear and logistic regression offered straightforward, interpretable models that worked well for many real-world problems. Support Vector Machines (SVMs) excelled at finding complex boundaries between categories using the "kernel trick"—imagine being able to untangle a bowl of spaghetti into straight lines by lifting it into a higher dimension. These algorithms formed the foundation of practical machine learning.

Consider a typical computer vision solution from 2005:

Example: Traditional Computer Vision Pipeline

```
1. Manual Feature Extraction
  - SIFT (Scale-Invariant Feature Transform)
  - HOG (Histogram of Oriented Gradients)
  - Gabor filters
2. Feature Selection/Engineering
3. "Shallow" Learning Model (e.g., SVM)
4. Post-processing
```

What made this era distinct was its hybrid approach: human-engineered features combined with statistical learning. They had strong mathematical foundations (researchers could prove why they worked). They performed well even with limited data. They were computationally efficient. They produced reliable, reproducible results.

Take the example of face detection, where the Viola-Jones algorithm (2001) achieved real-time performance using simple rectangular features and a cascade of classifiers. This algorithm powered digital camera face detection for nearly a decade.

1.3.5 Deep Learning Era

While Support Vector Machines excelled at finding complex boundaries between categories using mathematical transformations, deep learning took a radically different approach inspired by the human brain's architecture. Deep learning is built from layers of artificial neurons[4], where each layer learns to transform its input data into increasingly abstract representations. Imagine processing an image of a cat: the first layer might learn to detect simple edges and contrasts, the next layer combines these into basic shapes and textures, another layer might recognize whiskers and pointy ears, and the final layers assemble these features into the concept of "cat."

[4] Artificial Neurons: Basic computational units in neural networks that mimic biological neurons, taking multiple inputs, applying weights and biases, and producing an output signal through an activation function.

Unlike shallow learning methods that required humans to carefully engineer features, deep learning networks can automatically discover useful features directly from raw data. This ability to learn hierarchical representations, ranging from simple to complex and concrete to abstract, is what makes deep learning "deep," and it turned out to be a remarkably powerful approach for handling complex, real-world data like images, speech, and text.

In 2012, a deep neural network called AlexNet, shown in Figure 1.3, achieved a breakthrough in the ImageNet competition that would transform the field of machine learning. The challenge was formidable: correctly classify 1.2 million high-resolution images into 1,000 different categories. While previous approaches struggled with error rates above 25%, AlexNet[5] achieved a 15.3% error rate, dramatically outperforming all existing methods.

The success of AlexNet wasn't just a technical achievement; it was a watershed moment that demonstrated the practical viability of deep learning. It showed that with sufficient data, computational power, and architectural innovations, neural networks could outperform hand-engineered features and shallow learning methods that had dominated the field for decades. This single result triggered an explosion of research and applications in deep learning that continues to this day.

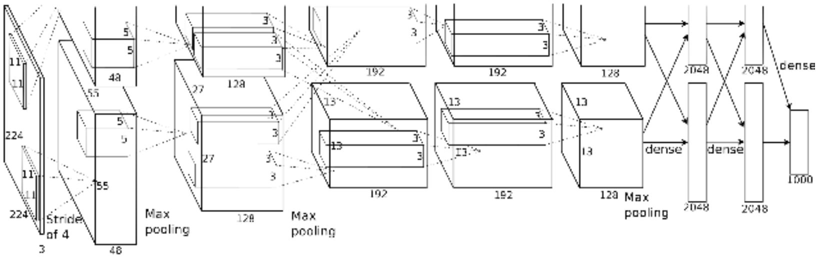


Figure 1.3: Deep neural network architecture for Alexnet. Source: Krizhevsky, Sutskever, and Hinton (2017a)

From this foundation, deep learning entered an era of unprecedented scale. By the late 2010s, companies like Google, Facebook, and OpenAI were training neural networks thousands of times larger than AlexNet. These massive models, often called "foundation models," took deep learning to new heights. GPT-3, released in 2020, contained 175 billion parameters[6]—imagine a student that could read through all of Wikipedia multiple times and learn patterns from every article. These models showed remarkable abilities: writing human-like text, engaging in conversation, generating images from descriptions, and even writing computer code. The key insight was simple but powerful: as we made neural networks bigger and fed them more data, they became capable of solving increasingly complex tasks. However, this scale brought unprecedented systems challenges: how do you efficiently train models that require thousands of GPUs working in parallel? How do you store and serve models that are hundreds of gigabytes in size? How do you handle the massive datasets needed for training?

The deep learning revolution of 2012 didn't emerge from nowhere, as it was founded on neural network research dating back to the 1950s. The story begins with Frank Rosenblatt's Perceptron in 1957, which captured the imagination of researchers by showing how a simple artificial neuron could learn to classify

[5] A breakthrough deep neural network from 2012 that won the ImageNet competition by a large margin and helped spark the deep learning revolution.

[6] Parameters: The adjustable values within a neural network that are modified during training, similar to how the brain's neural connections grow stronger as you learn a new skill. Having more parameters generally means that the model can learn more complex patterns.

patterns. While it could only handle linearly separable problems, a limitation that was dramatically highlighted by Minsky and Papert's 1969 book, "Perceptrons," it introduced the fundamental concept of trainable neural networks. The 1980s brought more important breakthroughs: Rumelhart, Hinton, and Williams introduced backpropagation in 1986, providing a systematic way to train multi-layer networks, while Yann LeCun demonstrated its practical application in recognizing handwritten digits using convolutional neural networks (CNNs)[7].

Yet these networks largely languished through the 1990s and 2000s, not because the ideas were wrong, but because they were ahead of their time. The field lacked three important ingredients: sufficient data to train complex networks, enough computational power to process this data, and the technical innovations needed to train very deep networks effectively.

! Convolutional Network Demo from 1989

🖥 Watch on YouTube

Watch on YouTube
Convolutional Net Demo

Scan with your phone
to watch the video

The field had to wait for the convergence of big data, better computing hardware, and algorithmic breakthroughs before deep learning's potential could be unlocked. This long gestation period helps explain why the 2012 ImageNet moment was less a sudden revolution and more the culmination of decades of accumulated research finally finding its moment. As we'll explore in the following sections, this evolution has led to two significant developments in the field. First, it has given rise to define the field of machine learning systems engineering, a discipline that teaches how to bridge the gap between theoretical advancements and practical implementation. Second, it has necessitated a more comprehensive definition of machine learning systems, one that encompasses not just algorithms, but also data and computing infrastructure. Today's challenges of scale echo many of the same fundamental questions about computation, data, and learning methods that researchers have grappled with since the field's inception, but now within a more complex and interconnected framework.

As AI progressed from symbolic reasoning to statistical learning and deep learning, its applications became increasingly ambitious and complex. This growth introduced challenges that extended beyond algorithms, necessitating a new focus: engineering entire systems capable of deploying and sustaining AI at scale. This gave rise to the discipline of Machine Learning Systems Engineering.

1.4 ML Systems Engineering

The story we've traced, from the early days of the Perceptron through the deep learning revolution, has largely been one of algorithmic breakthroughs. Each era brought new mathematical insights and modeling approaches that pushed the boundaries of what AI could achieve. But something important changed over the past decade: the success of AI systems became increasingly dependent not just on algorithmic innovations, but on sophisticated engineering.

[7] Convolutional Neural Network (CNN): A type of neural network specially designed for processing images, inspired by how the human visual system works. The "convolutional" part refers to how it scans images in small chunks, similar to how our eyes focus on different parts of a scene.

This shift mirrors the evolution of computer science and engineering in the late 1960s and early 1970s. During that period, as computing systems grew more complex, a new discipline emerged: Computer Engineering. This field bridged the gap between Electrical Engineering's hardware expertise and Computer Science's focus on algorithms and software. Computer Engineering arose because the challenges of designing and building complex computing systems required an integrated approach that neither discipline could fully address on its own.

Today, we're witnessing a similar transition in the field of AI. While Computer Science continues to push the boundaries of ML algorithms and Electrical Engineering advances specialized AI hardware, neither discipline fully addresses the engineering principles needed to deploy, optimize, and sustain ML systems at scale. This gap highlights the need for a new discipline: Machine Learning Systems Engineering.

There is no explicit definition of what this field is as such today, but it can be broadly defined as such:

Definition of Machine Learning Systems Engineering

> Machine Learning Systems Engineering (MLSysEng) is the discipline of designing, implementing, and operating artificially intelligent systems across computing scales—from resource-constrained embedded devices to warehouse-scale computers. This field integrates principles from engineering disciplines spanning hardware to software to create systems that are reliable, efficient, and optimized for their deployment context. It encompasses the complete lifecycle of AI applications: from requirements engineering and data collection through model development, system integration, deployment, monitoring, and maintenance. The field emphasizes engineering principles of systematic design, resource constraints, performance requirements, and operational reliability.

Let's consider space exploration. While astronauts venture into new frontiers and explore the vast unknowns of the universe, their discoveries are only possible because of the complex engineering systems supporting them, such as the rockets that lift them into space, the life support systems that keep them alive, and the communication networks that keep them connected to Earth. Similarly, while AI researchers push the boundaries of what's possible with learning algorithms, their breakthroughs only become practical reality through careful systems engineering. Modern AI systems need robust infrastructure to collect and manage data, powerful computing systems to train models, and reliable deployment platforms to serve millions of users.

This emergence of machine learning systems engineering as a important discipline reflects a broader reality: turning AI algorithms into real-world systems requires bridging the gap between theoretical possibilities and practical implementation. It's not enough to have a brilliant algorithm if you can't efficiently collect and process the data it needs, distribute its computation across hundreds of machines, serve it reliably to millions of users, or monitor its performance in production.

Understanding this interplay between algorithms and engineering has become fundamental for modern AI practitioners. While researchers continue to push the boundaries of what's algorithmically possible, engineers are tackling the complex challenge of making these algorithms work reliably and efficiently in the real world. This brings us to a fundamental question: what exactly is a machine learning system, and what makes it different from traditional software systems?

1.5 Defining ML Systems

There's no universally accepted, clear-cut textbook definition of a machine learning system. This ambiguity stems from the fact that different practitioners, researchers, and industries often refer to machine learning systems in varying contexts and with different scopes. Some might focus solely on the algorithmic aspects, while others might include the entire pipeline from data collection to model deployment. This loose usage of the term reflects the rapidly evolving and multidisciplinary nature of the field.

Given this diversity of perspectives, it is important to establish a clear and comprehensive definition that encompasses all these aspects. In this textbook, we take a holistic approach to machine learning systems, considering not just the algorithms but also the entire ecosystem in which they operate. Therefore, we define a machine learning system as follows:

> **Definition of a Machine Learning System**
>
> A machine learning system is an integrated computing system comprising three core components: (1) data that guides algorithmic behavior, (2) learning algorithms that extract patterns from this data, and (3) computing infrastructure that enables both the learning process (i.e., training) and the application of learned knowledge (i.e., inference/serving). Together, these components create a computing system capable of making predictions, generating content, or taking actions based on learned patterns.

The core of any machine learning system consists of three interrelated components, as illustrated in Figure 1.4: Models/Algorithms, Data, and Computing Infrastructure. These components form a triangular dependency where each element fundamentally shapes the possibilities of the others. The model architecture dictates both the computational demands for training and inference, as well as the volume and structure of data required for effective learning. The data's scale and complexity influence what infrastructure is needed for storage and processing, while simultaneously determining which model architectures are feasible. The infrastructure capabilities establish practical limits on both model scale and data processing capacity, creating a framework within which the other components must operate.

Each of these components serves a distinct but interconnected purpose:

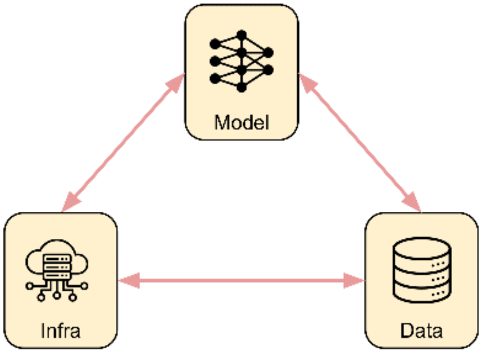


Figure 1.4: Machine learning systems involve algorithms, data, and computation, all intertwined together.

- **Algorithms**: Mathematical models and methods that learn patterns from data to make predictions or decisions
- **Data**: Processes and infrastructure for collecting, storing, processing, managing, and serving data for both training and inference.
- **Computing**: Hardware and software infrastructure that enables efficient training, serving, and operation of models at scale.

The interdependency of these components means no single element can function in isolation. The most sophisticated algorithm cannot learn without data or computing resources to run on. The largest datasets are useless without algorithms to extract patterns or infrastructure to process them. And the most powerful computing infrastructure serves no purpose without algorithms to execute or data to process.

To illustrate these relationships, we can draw an analogy to space exploration. Algorithm developers are like astronauts, who explore new frontiers and make discoveries. Data science teams function like mission control specialists, who ensure the constant flow of critical information and resources necessary to maintain the mission's operation. Computing infrastructure engineers are like rocket engineers—designing and building the systems that make the mission possible. Just as a space mission requires the seamless integration of astronauts, mission control, and rocket systems, a machine learning system demands the careful orchestration of algorithms, data, and computing infrastructure.

1.6 Lifecycle of ML Systems

Traditional software systems follow a predictable lifecycle where developers write explicit instructions for computers to execute. These systems are built on decades of established software engineering practices. Version control systems maintain precise histories of code changes. Continuous integration and deployment pipelines automate testing and release processes. Static analysis tools measure code quality and identify potential issues. This infrastructure enables reliable development, testing, and deployment of software systems, following well-defined principles of software engineering.

Machine learning systems represent a fundamental departure from this traditional paradigm. While traditional systems execute explicit programming logic, machine learning systems derive their behavior from patterns in data. This shift from code to data as the primary driver of system behavior introduces new complexities.

As illustrated in Figure 1.5, the ML lifecycle consists of interconnected stages from data collection through model monitoring, with feedback loops for continuous improvement when performance degrades or models need enhancement.

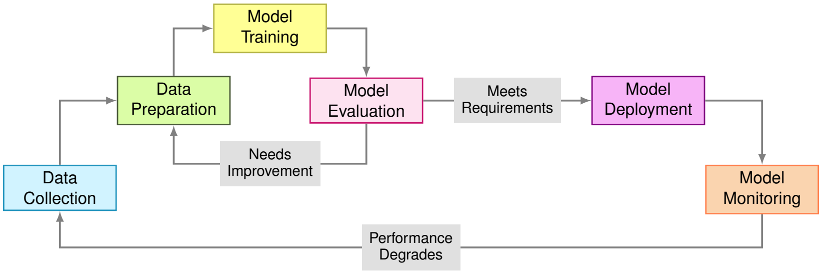


Figure 1.5: The typical lifecycle of a machine learning system.

Unlike source code, which changes only when developers modify it, data reflects the dynamic nature of the real world. Changes in data distributions can silently alter system behavior. Traditional software engineering tools, designed for deterministic code-based systems, prove insufficient for managing these data-dependent systems. For example, version control systems that excel at tracking discrete code changes struggle to manage large, evolving datasets. Testing frameworks designed for deterministic outputs must be adapted for probabilistic predictions. This data-dependent nature creates a more dynamic lifecycle, requiring continuous monitoring and adaptation to maintain system relevance as real-world data patterns evolve.

Understanding the machine learning system lifecycle requires examining its distinct stages. Each stage presents unique requirements from both learning and infrastructure perspectives. This dual consideration, of learning needs and systems support, is wildly important for building effective machine learning systems.

However, the various stages of the ML lifecycle in production are not isolated; they are, in fact, deeply interconnected. This interconnectedness can create either virtuous or vicious cycles. In a virtuous cycle, high-quality data enables effective learning, robust infrastructure supports efficient processing, and well-engineered systems facilitate the collection of even better data. However, in a vicious cycle, poor data quality undermines learning, inadequate infrastructure hampers processing, and system limitations prevent the improvement of data collection—each problem compounds the others.

1.7 ML Systems in the Wild

The complexity of managing machine learning systems becomes even more apparent when we consider the broad spectrum across which ML is deployed today. ML systems exist at vastly different scales and in diverse environments, each presenting unique challenges and constraints.

At one end of the spectrum, we have cloud-based ML systems running in massive data centers. These systems, like large language models or recommendation engines, process petabytes of data and serve millions of users simultaneously. They can leverage virtually unlimited computing resources but must manage enormous operational complexity and costs.

At the other end, we find TinyML systems running on microcontrollers and embedded devices. These systems must perform ML tasks with severe constraints on memory, computing power, and energy consumption. Imagine a smart home device, such as Alexa or Google Assistant, that must recognize voice commands using less power than a LED bulb, or a sensor that must detect anomalies while running on a battery for months or even years.

Between these extremes, we find a rich variety of ML systems adapted for different contexts. Edge ML systems bring computation closer to data sources, reducing latency and bandwidth requirements while managing local computing resources. Mobile ML systems must balance sophisticated capabilities with battery life and processor limitations on smartphones and tablets. Enterprise ML systems often operate within specific business constraints, focusing on particular tasks while integrating with existing infrastructure. Some organizations employ hybrid approaches, distributing ML capabilities across multiple tiers to balance various requirements.

1.8 ML Systems Impact on Lifecycle

The diversity of ML systems across the spectrum represents a complex interplay of requirements, constraints, and trade-offs. These decisions fundamentally impact every stage of the ML lifecycle we discussed earlier, from data collection to continuous operation.

Performance requirements often drive initial architectural decisions. Latency-sensitive applications, like autonomous vehicles or real-time fraud detection, might require edge or embedded architectures despite their resource constraints. Conversely, applications requiring massive computational power for training, such as large language models, naturally gravitate toward centralized cloud architectures. However, raw performance is just one consideration in a complex decision space.

Resource management varies dramatically across architectures. Cloud systems must optimize for cost efficiency at scale—balancing expensive GPU clusters, storage systems, and network bandwidth. Edge systems face fixed resource limits and must carefully manage local compute and storage. Mobile and embedded systems operate under the strictest constraints, where every byte of memory and milliwatt of power matters. These resource considerations directly influence both model design and system architecture.

Operational complexity increases with system distribution. While centralized cloud architectures benefit from mature deployment tools and managed services, edge and hybrid systems must handle the complexity of distributed system management. This complexity manifests throughout the ML lifecycle—from data collection and version control to model deployment and monitoring. This operational complexity can compound over time if not carefully managed.

Data considerations often introduce competing pressures. Privacy requirements or data sovereignty regulations might push toward edge or embedded architectures, while the need for large-scale training data might favor cloud approaches. The velocity and volume of data also influence architectural choices—real-time sensor data might require edge processing to manage bandwidth, while batch analytics might be better suited to cloud processing.

Evolution and maintenance requirements must be considered from the start. Cloud architectures offer flexibility for system evolution but can incur significant ongoing costs. Edge and embedded systems might be harder to update but could offer lower operational overhead. The continuous cycle of ML systems we discussed earlier becomes particularly challenging in distributed architectures, where updating models and maintaining system health requires careful orchestration across multiple tiers.

These trade-offs are rarely simple binary choices. Modern ML systems often adopt hybrid approaches, carefully balancing these considerations based on specific use cases and constraints. The key is understanding how these decisions will impact the system throughout its lifecycle, from initial development through continuous operation and evolution.

1.8.1 Emerging Trends

The landscape of machine learning systems is evolving rapidly, with innovations happening from user-facing applications down to core infrastructure. These changes are reshaping how we design and deploy ML systems.

1.8.1.1 Application-Level Innovation

The rise of agentic systems marks a profound shift from traditional reactive ML systems that simply made predictions based on input data. Modern applications can now take actions, learn from outcomes, and adapt their behavior accordingly through multi-agent systems[8] and advanced planning algorithms. These autonomous agents can plan, reason, and execute complex tasks, introducing new requirements for decision-making frameworks and safety constraints.

This increased sophistication extends to operational intelligence. Applications will likely incorporate sophisticated self-monitoring, automated resource management, and adaptive deployment strategies. They can automatically handle data distribution shifts, model updates, and system optimization, marking a significant advance in autonomous operation.

1.8.1.2 System Architecture Evolution

Supporting these advanced applications requires fundamental changes in the underlying system architecture. Integration frameworks are evolving to handle increasingly complex interactions between ML systems and broader technology ecosystems. Modern ML systems must seamlessly connect with existing software, process diverse data sources, and operate across organizational boundaries, driving new approaches to system design.

Resource efficiency has become a central architectural concern as ML systems scale. Innovation in model compression and efficient training techniques is

[8] Multi-Agent System: A computational system where multiple intelligent agents interact within an environment, each pursuing their own objectives while potentially cooperating or competing with other agents.

being driven by both environmental and economic factors. Future architectures must carefully balance the pursuit of more powerful models against growing sustainability concerns.

At the infrastructure level, new hardware is reshaping deployment possibilities. Specialized AI accelerators are emerging across the spectrum—from powerful data center chips to efficient edge processors[9] to tiny neural processing units in mobile devices. This heterogeneous computing landscape enables dynamic model distribution across tiers based on computing capabilities and conditions, blurring traditional boundaries between cloud, edge, and embedded systems.

These trends are creating ML systems that are more capable and efficient while managing increasing complexity. Success in this evolving landscape requires understanding how application requirements flow down to infrastructure decisions, ensuring systems can grow sustainably while delivering increasingly sophisticated capabilities.

1.9 Practical Applications

The diverse architectures and scales of ML systems demonstrate their potential to revolutionize industries. By examining real-world applications, we can see how these systems address practical challenges and drive innovation. Their ability to operate effectively across varying scales and environments has already led to significant changes in numerous sectors. This section highlights examples where theoretical concepts and practical considerations converge to produce tangible, impactful results.

1.9.1 FarmBeats: ML in Agriculture

FarmBeats, a project developed by Microsoft Research, shown in Figure 1.6 is a significant advancement in the application of machine learning to agriculture. This system aims to increase farm productivity and reduce costs by leveraging AI and IoT technologies. FarmBeats exemplifies how edge and embedded ML systems can be deployed in challenging, real-world environments to solve practical problems. By bringing ML capabilities directly to the farm, FarmBeats demonstrates the potential of distributed AI systems in transforming traditional industries.

1.9.1.1 Data Considerations

The data ecosystem in FarmBeats is diverse and distributed. Sensors deployed across fields collect real-time data on soil moisture, temperature, and nutrient levels. Drones equipped with multispectral cameras capture high-resolution imagery of crops, providing insights into plant health and growth patterns. Weather stations contribute local climate data, while historical farming records offer context for long-term trends. The challenge lies not just in collecting this heterogeneous data, but in managing its flow from dispersed, often remote locations with limited connectivity. FarmBeats employs innovative data transmission techniques, such as using TV white spaces (unused broadcasting frequencies) to extend internet connectivity to far-flung sensors. This approach

[9] Edge Processor: A specialized computing device designed to perform AI computations close to where data is generated, optimized for low latency and energy efficiency rather than raw computing power.

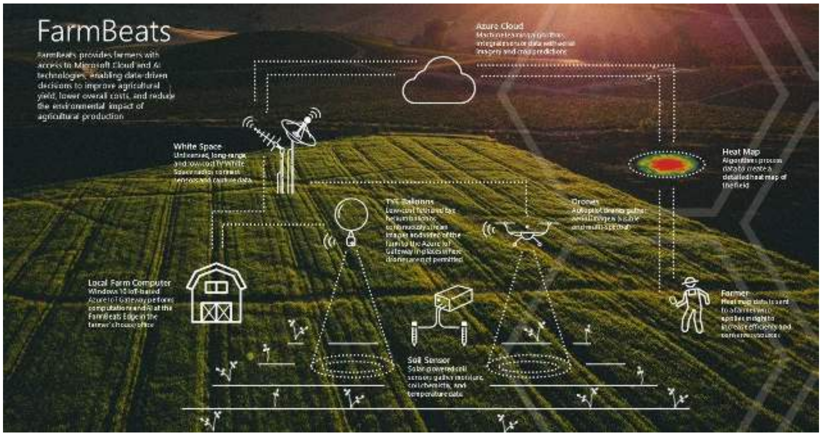

FarmBeats
FarmBeats provides farmers with access to Microsoft Cloud and AI technologies, enabling data-driven decisions to improve agricultural yield, lower overall costs, and reduce the environmental impact of agricultural production
Azure Cloud
Machine learning algorithms integrate sensor data with aerial imagery and map predictions
White Space
Unlicensed, long-range, and low-cost TV White Space radios connect sensors and capture data.
Heat Map
Algorithms process data to create a detailed heat map of the field
TVS Balloons
Low-cost tethered helium balloons continuously stream images and video of the farm to the Azure IoT Gateway in places where drones are not permitted
Drones
Autopilot drones gather aerial imagery (visible and multi-spectral)
Local Farm Computer
Windows 10 IoT-based Azure IoT Gateway performs computation and AI at the FarmBeats Edge in the farmer's house/office
Farmer
Heat map data is sent to a farmer who applies insights to increase efficiency and conserve resources
Soil Sensor
Solar-powered soil sensors gather moisture, soil chemistry, and temperature data

Figure 1.6: Microsoft FarmBeats: AI, Edge & IoT for Agriculture.

to data collection and transmission embodies the principles of edge computing we discussed earlier, where data processing begins at the source to reduce bandwidth requirements and enable real-time decision making.

1.9.1.2 Algorithmic Considerations

FarmBeats uses a variety of ML algorithms tailored to agricultural applications. For soil moisture prediction, it uses temporal neural networks that can capture the complex dynamics of water movement in soil. Computer vision algorithms process drone imagery to detect crop stress, pest infestations, and yield estimates. These models must be robust to noisy data and capable of operating with limited computational resources. Machine learning methods such as "transfer learning" allow models to learn on data-rich farms to be adapted for use in areas with limited historical data. The system also incorporates a mixture of methods that combine outputs from multiple algorithms to improve prediction accuracy and reliability. A key challenge FarmBeats addresses is model personalization, adapting general models to the specific conditions of individual farms. These conditions may include unique soil compositions, microclimates, and farming practices.

1.9.1.3 Infrastructure Considerations

FarmBeats exemplifies the edge computing paradigm we explored in our discussion of the ML system spectrum. At the lowest level, embedded ML models run directly on IoT devices and sensors, performing basic data filtering and anomaly detection. Edge devices, such as ruggedized field gateways, aggregate data from multiple sensors and run more complex models for local decision-making. These edge devices operate in challenging conditions, requiring robust hardware designs and efficient power management to function reliably in remote agricultural settings. The system employs a hierarchical architecture, with more computationally intensive tasks offloaded to on-premises servers or the cloud. This tiered approach allows FarmBeats to balance the need for real-time processing with the benefits of centralized data analysis and model training.

The infrastructure also includes mechanisms for over-the-air model updates, ensuring that edge devices can receive improved models as more data becomes available and algorithms are refined.

1.9.1.4 Future Implications

FarmBeats shows how ML systems can be deployed in resource-constrained, real-world environments to drive significant improvements in traditional industries. By providing farmers with AI-driven insights, the system has shown potential to increase crop yields, reduce water usage, and optimize resource allocation. Looking forward, the FarmBeats approach could be extended to address global challenges in food security and sustainable agriculture. The success of this system also highlights the growing importance of edge and embedded ML in IoT applications, where bringing intelligence closer to the data source can lead to more responsive, efficient, and scalable solutions. As edge computing capabilities continue to advance, we can expect to see similar distributed ML architectures applied to other domains, from smart cities to environmental monitoring.

1.9.2 AlphaFold: Scientific ML

AlphaFold, developed by DeepMind, is a landmark achievement in the application of machine learning to complex scientific problems. This AI system is designed to predict the three-dimensional structure of proteins, as shown in Figure 1.7, from their amino acid sequences, a challenge known as the "protein folding problem" that has puzzled scientists for decades. AlphaFold's success demonstrates how large-scale ML systems can accelerate scientific discovery and potentially revolutionize fields like structural biology and drug design. This case study exemplifies the use of advanced ML techniques and massive computational resources to tackle problems at the frontiers of science.

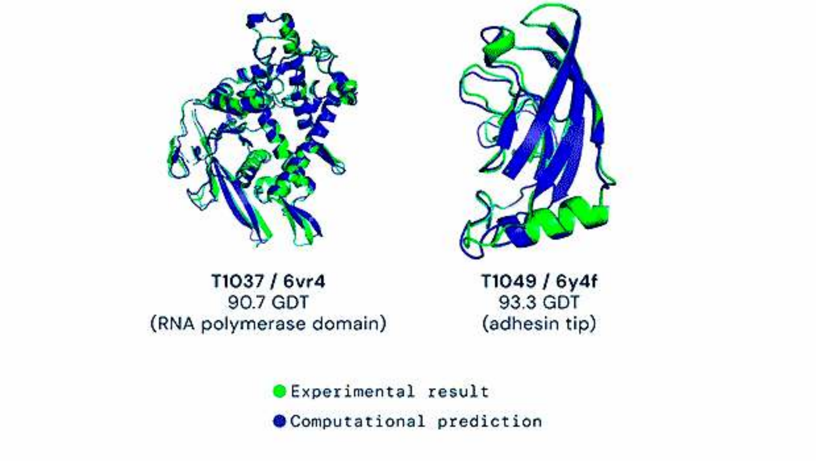


Figure 1.7: Examples of protein targets within the free modeling category. Source: Google DeepMind

1.9.2.1 Data Considerations

The data underpinning AlphaFold's success is vast and multifaceted. The primary dataset is the Protein Data Bank (PDB), which contains the experimentally determined structures of over 180,000 proteins. This is complemented by databases of protein sequences, which number in the hundreds of millions. AlphaFold also utilizes evolutionary data in the form of multiple sequence alignments (MSAs), which provide insights into the conservation patterns of amino acids across related proteins. The challenge lies not just in the volume of data, but in its quality and representation. Experimental protein structures can contain errors or be incomplete, requiring sophisticated data cleaning and validation processes. Moreover, the representation of protein structures and sequences in a form amenable to machine learning is a significant challenge in itself. AlphaFold's data pipeline involves complex preprocessing steps to convert raw sequence and structural data into meaningful features that capture the physical and chemical properties relevant to protein folding.

1.9.2.2 Algorithmic Considerations

AlphaFold's algorithmic approach represents a tour de force in the application of deep learning to scientific problems. At its core, AlphaFold uses a novel neural network architecture that combines with techniques from computational biology. The model learns to predict inter-residue distances and torsion angles, which are then used to construct a full 3D protein structure. A key innovation is the use of "equivariant attention" layers that respect the symmetries inherent in protein structures. The learning process involves multiple stages, including initial "pretraining" on a large corpus of protein sequences, followed by fine-tuning on known structures. AlphaFold also incorporates domain knowledge in the form of physics-based constraints and scoring functions, creating a hybrid system that leverages both data-driven learning and scientific prior knowledge. The model's ability to generate accurate confidence estimates for its predictions is crucial, allowing researchers to assess the reliability of the predicted structures.

1.9.2.3 Infrastructure Considerations

The computational demands of AlphaFold epitomize the challenges of large-scale scientific ML systems. Training the model requires massive parallel computing resources, leveraging clusters of GPUs or TPUs (Tensor Processing Units) in a distributed computing environment. DeepMind utilized Google's cloud infrastructure, with the final version of AlphaFold trained on 128 TPUv3 cores for several weeks. The inference process, while less computationally intensive than training, still requires significant resources, especially when predicting structures for large proteins or processing many proteins in parallel. To make AlphaFold more accessible to the scientific community, DeepMind has collaborated with the European Bioinformatics Institute to create a public database of predicted protein structures, which itself represents a substantial computing and data management challenge. This infrastructure allows researchers worldwide to access AlphaFold's predictions without needing to run the model themselves, demonstrating how centralized, high-performance computing resources can be leveraged to democratize access to advanced ML capabilities.

1.9.2.4 Future Implications

AlphaFold's impact on structural biology has been profound, with the potential to accelerate research in areas ranging from fundamental biology to drug discovery. By providing accurate structural predictions for proteins that have resisted experimental methods, AlphaFold opens new avenues for understanding disease mechanisms and designing targeted therapies. The success of AlphaFold also serves as a powerful demonstration of how ML can be applied to other complex scientific problems, potentially leading to breakthroughs in fields like materials science or climate modeling. However, it also raises important questions about the role of AI in scientific discovery and the changing nature of scientific inquiry in the age of large-scale ML systems. As we look to the future, the AlphaFold approach suggests a new paradigm for scientific ML, where massive computational resources are combined with domain-specific knowledge to push the boundaries of human understanding.

1.9.3 Autonomous Vehicles and ML

Waymo, a subsidiary of Alphabet Inc., stands at the forefront of autonomous vehicle technology, representing one of the most ambitious applications of machine learning systems to date. Evolving from the Google Self-Driving Car Project initiated in 2009, Waymo's approach to autonomous driving exemplifies how ML systems can span the entire spectrum from embedded systems to cloud infrastructure. This case study demonstrates the practical implementation of complex ML systems in a safety-critical, real-world environment, integrating real-time decision-making with long-term learning and adaptation.

1.9.3.1 Data Considerations

The data ecosystem underpinning Waymo's technology is vast and dynamic. Each vehicle serves as a roving data center, its sensor suite, which comprises LiDAR, radar, and high-resolution cameras, generating approximately one terabyte of data per hour of driving. This real-world data is complemented by an even more extensive simulated dataset, with Waymo's vehicles having traversed over 20 billion miles in simulation and more than 20 million miles on public roads. The challenge lies not just in the volume of data, but in its heterogeneity and the need for real-time processing. Waymo must handle both structured (e.g., GPS coordinates) and unstructured data (e.g., camera images) simultaneously. The data pipeline spans from edge processing on the vehicle itself to massive cloud-based storage and processing systems. Sophisticated data cleaning and validation processes are necessary, given the safety-critical nature of the application. Moreover, the representation of the vehicle's environment in a form amenable to machine learning presents significant challenges, requiring complex preprocessing to convert raw sensor data into meaningful features that capture the dynamics of traffic scenarios.

1.9.3.2 Algorithmic Considerations

Waymo's ML stack represents a sophisticated ensemble of algorithms tailored to the multifaceted challenge of autonomous driving. The perception system employs deep learning techniques, including convolutional neural networks, to process visual data for object detection and tracking. Prediction models, needed for anticipating the behavior of other road users, leverage recurrent neural networks (RNNs)[10] to understand temporal sequences. Waymo has developed custom ML models like VectorNet for predicting vehicle trajectories. The planning and decision-making systems may incorporate reinforcement learning or imitation learning techniques to navigate complex traffic scenarios. A key innovation in Waymo's approach is the integration of these diverse models into a coherent system capable of real-time operation. The ML models must also be interpretable to some degree, as understanding the reasoning behind a vehicle's decisions is vital for safety and regulatory compliance. Waymo's learning process involves continuous refinement based on real-world driving experiences and extensive simulation, creating a feedback loop that constantly improves the system's performance.

1.9.3.3 Infrastructure Considerations

The computing infrastructure supporting Waymo's autonomous vehicles epitomizes the challenges of deploying ML systems across the full spectrum from edge to cloud. Each vehicle is equipped with a custom-designed compute platform capable of processing sensor data and making decisions in real-time, often leveraging specialized hardware like GPUs or tensor processing units (TPUs)[11]. This edge computing is complemented by extensive use of cloud infrastructure, leveraging the power of Google's data centers for training models, running large-scale simulations, and performing fleet-wide learning. The connectivity between these tiers is critical, with vehicles requiring reliable, high-bandwidth communication for real-time updates and data uploading. Waymo's infrastructure must be designed for robustness and fault tolerance, ensuring safe operation even in the face of hardware failures or network disruptions. The scale of Waymo's operation presents significant challenges in data management, model deployment, and system monitoring across a geographically distributed fleet of vehicles.

1.9.3.4 Future Implications

Waymo's impact extends beyond technological advancement, potentially revolutionizing transportation, urban planning, and numerous aspects of daily life. The launch of Waymo One, a commercial ride-hailing service using autonomous vehicles in Phoenix, Arizona, represents a significant milestone in the practical deployment of AI systems in safety-critical applications. Waymo's progress has broader implications for the development of robust, real-world AI systems, driving innovations in sensor technology, edge computing, and AI safety that have applications far beyond the automotive industry. However, it also raises important questions about liability, ethics, and the interaction between AI systems and human society. As Waymo continues to expand its

[10] Recurrent Neural Network (RNN): A type of neural network specifically designed to handle sequential data by maintaining an internal memory state that allows it to learn patterns across time, making it particularly useful for tasks like language processing and time series prediction.

[11] Tensor Processing Unit (TPU): A specialized AI accelerator chip designed by Google specifically for neural network machine learning, particularly efficient at matrix operations common in deep learning workloads.

operations and explore applications in trucking and last-mile delivery, it serves as an important test bed for advanced ML systems, driving progress in areas such as continual learning, robust perception, and human-AI interaction. The Waymo case study underscores both the tremendous potential of ML systems to transform industries and the complex challenges involved in deploying AI in the real world.

1.10 Challenges in ML Systems

Building and deploying machine learning systems presents unique challenges that go beyond traditional software development. These challenges help explain why creating effective ML systems is about more than just choosing the right algorithm or collecting enough data. Let's explore the key areas where ML practitioners face significant hurdles.

1.10.1 Data-Related Challenges

The foundation of any ML system is its data, and managing this data introduces several fundamental challenges. First, there's the basic question of data quality, as real-world data is often messy and inconsistent. Imagine a healthcare application that needs to process patient records from different hospitals. Each hospital might record information differently, use different units of measurement, or have different standards for what data to collect. Some records might have missing information, while others might contain errors or inconsistencies that need to be cleaned up before the data can be useful.

As ML systems grow, they often need to handle increasingly large amounts of data. A video streaming service like Netflix, for example, needs to process billions of viewer interactions to power its recommendation system. This scale introduces new challenges in how to store, process, and manage such large datasets efficiently.

Another critical challenge is how data changes over time. This phenomenon, known as "data drift"[12], occurs when the patterns in new data begin to differ from the patterns the system originally learned from. For example, many predictive models struggled during the COVID-19 pandemic because consumer behavior changed so dramatically that historical patterns became less relevant. ML systems need ways to detect when this happens and adapt accordingly.

1.10.2 Model-Related Challenges

Creating and maintaining the ML models themselves presents another set of challenges. Modern ML models, particularly in deep learning, can be extremely complex. Consider a language model like GPT-3, which has hundreds of billions of parameters that need to be optimized through backpropagation[13]. This complexity creates practical challenges: these models require enormous computing power to train and run, making it difficult to deploy them in situations with limited resources, like on mobile phones or IoT devices.

Training these models effectively is itself a significant challenge. Unlike traditional programming where we write explicit instructions, ML models learn from examples through techniques like transfer learning[14]. This learning

[12] Data Drift: The gradual change in the statistical properties of the target variable (what the model is trying to predict) over time, which can degrade model performance if not properly monitored and addressed.

[13] Backpropagation: The primary algorithm used to train neural networks, which calculates how each parameter in the network should be adjusted to minimize prediction errors by propagating error gradients backward through the network layers.

[14] Transfer Learning: A machine learning method where a model developed for one task is reused as the starting point for a model on a second task, significantly reducing the amount of training data and computation required.

process involves many choices: How should we structure the model? How long should we train it? How can we tell if it's learning the right things? Making these decisions often requires both technical expertise and considerable trial and error.

A particularly important challenge is ensuring that models work well in real-world conditions. A model might perform excellently on its training data but fail when faced with slightly different situations in the real world. This gap between training performance and real-world performance is a central challenge in machine learning, especially for critical applications like autonomous vehicles or medical diagnosis systems.

1.10.3 System-Related Challenges

Getting ML systems to work reliably in the real world introduces its own set of challenges. Unlike traditional software that follows fixed rules, ML systems need to handle uncertainty and variability in their inputs and outputs. They also typically need both training systems (for learning from data) and serving systems (for making predictions), each with different requirements and constraints.

Consider a company building a speech recognition system. They need infrastructure to collect and store audio data, systems to train models on this data, and then separate systems to actually process users' speech in real-time. Each part of this pipeline needs to work reliably and efficiently, and all the parts need to work together seamlessly.

These systems also need constant monitoring and updating. How do we know if the system is working correctly? How do we update models without interrupting service? How do we handle errors or unexpected inputs? These operational challenges become particularly complex when ML systems are serving millions of users.

1.10.4 Ethical Considerations

As ML systems become more prevalent in our daily lives, their broader impacts on society become increasingly important to consider. One major concern is fairness, as ML systems can sometimes learn to make decisions that discriminate against certain groups of people. This often happens unintentionally, as the systems pick up biases present in their training data. For example, a job application screening system might inadvertently learn to favor certain demographics if those groups were historically more likely to be hired.

Another important consideration is transparency. Many modern ML models, particularly deep learning models, work as "black boxes"—while they can make predictions, it's often difficult to understand how they arrived at their decisions. This becomes particularly problematic when ML systems are making important decisions about people's lives, such as in healthcare or financial services.

Privacy is also a major concern. ML systems often need large amounts of data to work effectively, but this data might contain sensitive personal information. How do we balance the need for data with the need to protect individual privacy? How do we ensure that models don't inadvertently memorize and reveal private information through inference attacks[15]? These challenges aren't

[15] Inference Attack: A technique where an adversary attempts to extract sensitive information about the training data by making careful queries to a trained model, exploiting patterns the model may have inadvertently memorized during training.

merely technical problems to be solved, but ongoing considerations that shape how we approach ML system design and deployment.

These challenges aren't merely technical problems to be solved, but ongoing considerations that shape how we approach ML system design and deployment. Throughout this book, we'll explore these challenges in detail and examine strategies for addressing them effectively.

1.11 Looking Ahead

As we look to the future of machine learning systems, several exciting trends are shaping the field. These developments promise to both solve existing challenges and open new possibilities for what ML systems can achieve.

One of the most significant trends is the democratization of AI technology. Just as personal computers transformed computing from specialized mainframes to everyday tools, ML systems are becoming more accessible to developers and organizations of all sizes. Cloud providers now offer pre-trained models and automated ML platforms that reduce the expertise needed to deploy AI solutions. This democratization is enabling new applications across industries, from small businesses using AI for customer service to researchers applying ML to previously intractable problems.

As concerns about computational costs and environmental impact grow, there's an increasing focus on making ML systems more efficient. Researchers are developing new techniques for training models with less data and computing power. Innovation in specialized hardware, from improved GPUs to custom AI chips, is making ML systems faster and more energy-efficient. These advances could make sophisticated AI capabilities available on more devices, from smartphones to IoT sensors.

Perhaps the most transformative trend is the development of more autonomous ML systems that can adapt and improve themselves. These systems are beginning to handle their own maintenance tasks, such as detecting when they need retraining, automatically finding and correcting errors, and optimizing their own performance. This automation could dramatically reduce the operational overhead of running ML systems while improving their reliability.

While these trends are promising, it's important to recognize the field's limitations. Creating truly artificial general intelligence remains a distant goal. Current ML systems excel at specific tasks but lack the flexibility and understanding that humans take for granted. Challenges around bias, transparency, and privacy continue to require careful consideration. As ML systems become more prevalent, addressing these limitations while leveraging new capabilities will be crucial.

1.12 Book Structure and Learning Path

This book is designed to guide you from understanding the fundamentals of ML systems to effectively designing and implementing them. To address the complexities and challenges of Machine Learning Systems engineering, we've organized the content around five fundamental pillars that encompass the

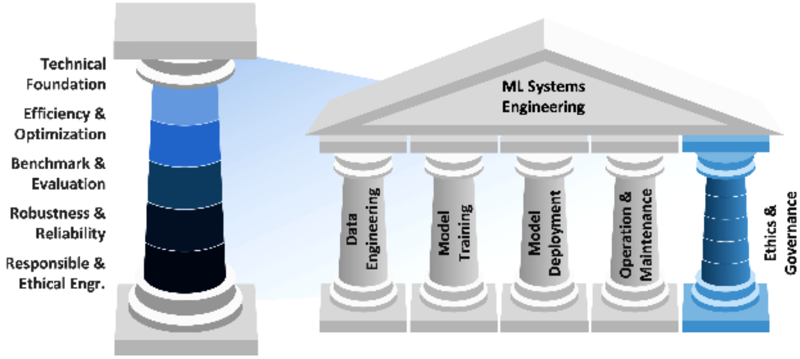


Figure 1.8: Overview of the five fundamental system pillars of Machine Learning Systems engineering.

lifecycle of ML systems. These pillars provide a framework for understanding, developing, and maintaining robust ML systems.

As illustrated in Figure 1.8, the five pillars central to the framework are:

- **Data**: Emphasizing data engineering and foundational principles critical to how AI operates in relation to data.
- **Training**: Exploring the methodologies for AI training, focusing on efficiency, optimization, and acceleration techniques to enhance model performance.
- **Deployment**: Encompassing benchmarks, on-device learning strategies, and machine learning operations to ensure effective model application.
- **Operations**: Highlighting the maintenance challenges unique to machine learning systems, which require specialized approaches distinct from traditional engineering systems.
- **Ethics & Governance**: Addressing concerns such as security, privacy, responsible AI practices, and the broader societal implications of AI technologies.

Each pillar represents a critical phase in the lifecycle of ML systems and is composed of foundational elements that build upon each other. This structure ensures a comprehensive understanding of MLSE, from basic principles to advanced applications and ethical considerations.

For more detailed information about the book's overview, contents, learning outcomes, target audience, prerequisites, and navigation guide, please refer to the About the Book section. There, you'll also find valuable details about our learning community and how to maximize your experience with this resource.

Chapter 2

ML Systems

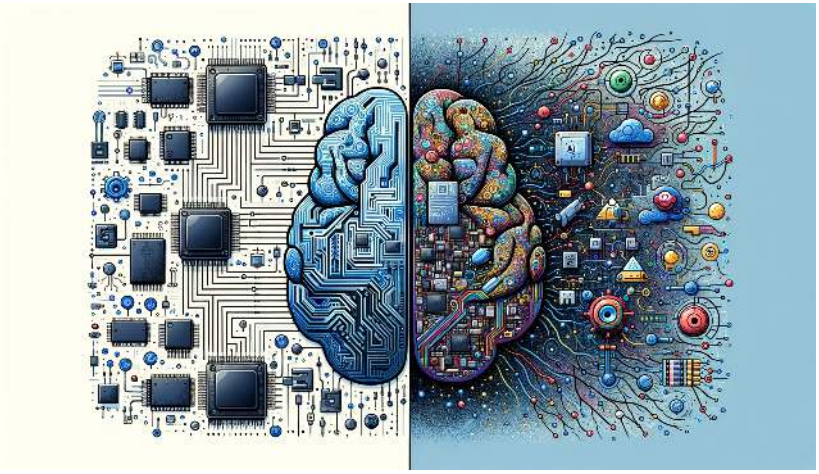


Figure 2.1: *DALL·E 3 Prompt: Illustration in a rectangular format depicting the merger of embedded systems with Embedded AI. The left half of the image portrays traditional embedded systems, including microcontrollers and processors, detailed and precise. The right half showcases the world of artificial intelligence, with abstract representations of machine learning models, neurons, and data flow. The two halves are distinctly separated, emphasizing the individual significance of embedded tech and AI, but they come together in harmony at the center.*

Purpose

How do the diverse environments where machine learning operates shape the fundamental nature of these systems, and what drives their widespread deployment across computing platforms?

The deployment of machine learning systems across varied computing environments reveals essential insights into the relationship between theoretical principles and practical implementation. Each computing environment, from large-scale distributed systems to resource-constrained devices, introduces distinct requirements that influence both system architecture and algorithmic approaches. Understanding these relationships reveals core engineering principles that govern the design of machine learning systems. This understanding provides a foundation for examining how theoretical concepts translate into practical implementations, and how system designs adapt to meet diverse computational, memory, and energy constraints.

Learning Objectives

- Understand the key characteristics and differences between Cloud ML, Edge ML, Mobile ML, and Tiny ML systems.
- Analyze the benefits and challenges associated with each ML paradigm.
- Explore real-world applications and use cases for Cloud ML, Edge ML, Mobile ML, and Tiny ML.
- Compare the performance aspects of each ML approach, including latency, privacy, and resource utilization.
- Examine the evolving landscape of ML systems and potential future developments.

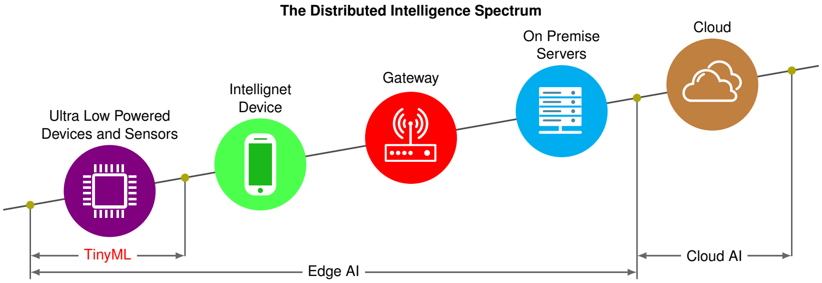
2.1 Overview

Modern machine learning systems span a spectrum of deployment options, each with its own set of characteristics and use cases. At one end, we have cloud-based ML, which leverages powerful centralized computing resources for complex, data-intensive tasks. Moving along the spectrum, we encounter edge ML, which brings computation closer to the data source for reduced latency and improved privacy. Mobile ML further extends these capabilities to smartphones and tablets, while at the far end, we find Tiny ML, which enables machine learning on extremely low-power devices with severe memory and processing constraints.

This spectrum of deployment can be visualized like Earth's geological features, each operating at different scales in our computational landscape. Cloud ML systems operate like continents, processing vast amounts of data across interconnected centers; Edge ML exists where these continental powers meet the sea, creating dynamic coastlines where computation flows into local waters; Mobile ML moves through these waters like ocean currents, carrying computing power across the digital seas; and where these currents meet the physical world, TinyML systems rise like islands, each a precise point of intelligence in the vast computational ocean.

Figure 2.2 illustrates the spectrum of distributed intelligence across these approaches, providing a visual comparison of their characteristics. We will examine the unique characteristics, advantages, and challenges of each approach, as depicted in the figure. Additionally, we will discuss the emerging trends and technologies that are shaping the future of machine learning deployment, considering how they might influence the balance between these three paradigms.

To better understand the dramatic differences between these ML deployment options, Table 2.1 provides examples of representative hardware platforms for each category. These examples illustrate the vast range of computational resources, power requirements, and cost considerations across the ML systems spectrum. As we explore each paradigm in detail, you can refer back to


The Distributed Intelligence Spectrum
Ultra Low Powered Devices and Sensors
Intellignet Device
Gateway
On Premise Servers
Cloud
TinyML
Edge AI
Cloud AI

Source: ABI Research: TinyML

Figure 2.2: Cloud vs. Edge vs. Mobile vs. Tiny ML: The Spectrum of Distributed Intelligence. Source: ABI Research – Tiny ML.

these concrete examples to better understand the practical implications of each approach.

Table 2.1: Representative hardware platforms across the ML systems spectrum, showing typical specifications and capabilities for each category.

Category	Example Device	Processor	Memory	Storage	Power	Price Range	Example Models/Tasks
Cloud ML	NVIDIA DGX A100	8x NVIDIA A100 GPUs (40 GB/80 GB)	1 TB System RAM	15 TB NVMe SSD	6.5 kW	$200 K+	Large language models (GPT-3), real-time video processing
	Google TPU v4 Pod	4096 TPU v4 chips	128 TB+	Networked storage	~MW	Pay-per-use	Training foundation models, large-scale ML research
Edge ML	NVIDIA Jetson AGX Orin	12-core Arm® Cortex®-A78AE, NVIDIA Ampere GPU	32 GB LPDDR5	64GB eMMC	15-60 W	$899	Computer vision, robotics, autonomous systems
	Intel NUC 12 Pro	Intel Core i7-1260P, Intel Iris Xe	32 GB DDR4	1 TB SSD	28 W	$750	Edge AI servers, industrial automation
Mobile ML	iPhone 15 Pro	A17 Pro (6-core CPU, 6-core GPU)	8 GB RAM	128 GB-1 TB	3-5 W	$999+	Face ID, computational photography, voice recognition
Tiny ML	Arduino Nano 33 BLE Sense	Arm Cortex-M4 @ 64 MHz	256 KB RAM	1 MB Flash	0.02-0.04 W	$35	Gesture recognition, voice detection
	ESP32-CAM	Dual-core @ 240MHz	520 KB RAM	4 MB Flash	0.05-0.25 W	$10	Image classification, motion detection

The evolution of machine learning systems can be seen as a progression from centralized to increasingly distributed and specialized computing paradigms:

Cloud ML: Initially, ML was predominantly cloud-based. Powerful, scalable servers in data centers are used to train and run large ML models. This approach leverages vast computational resources and storage capacities, enabling the development of complex models trained on massive datasets. Cloud ML excels at tasks requiring extensive processing power, distributed training of large models, and is ideal for applications where real-time responsiveness isn't critical. Popular platforms like AWS SageMaker, Google Cloud AI, and Azure ML offer flexible, scalable solutions for model development, training, and deployment. Cloud ML can handle models with billions of parameters, training on petabytes of data, but may incur latencies of 100-500 ms for online inference due to network delays.

Edge ML: As the need for real-time, low-latency processing grew, Edge ML emerged. This paradigm brings inference capabilities closer to the data source, typically on edge devices such as industrial gateways, smart cameras, autonomous vehicles, or IoT hubs. Edge ML reduces latency (often to less than 50 ms), enhances privacy by keeping data local, and can operate with intermittent cloud connectivity. It's particularly useful for applications requiring quick responses or handling sensitive data in industrial or enterprise settings. Frameworks like NVIDIA Jetson or Google's Edge TPU enable powerful ML capabilities on edge devices. Edge ML plays a crucial role in IoT ecosystems, enabling real-time decision making and reducing bandwidth usage by processing data locally.

Mobile ML: Building on edge computing concepts, Mobile ML focuses on leveraging the computational capabilities of smartphones and tablets. This approach enables personalized, responsive applications while reducing reliance on constant network connectivity. Mobile ML offers a balance between the power of edge computing and the ubiquity of personal devices. It utilizes on-device sensors (e.g., cameras, GPS, accelerometers) for unique ML applications. Frameworks like TensorFlow Lite and Core ML allow developers to deploy optimized models on mobile devices, with inference times often under 30 ms for common tasks. Mobile ML enhances privacy by keeping personal data on the device and can operate offline, but must balance model performance with device resource constraints (typically 4-8 GB RAM, 100-200 GB storage).

Tiny ML: The latest development in this progression is Tiny ML, which enables ML models to run on extremely resource-constrained microcontrollers and small embedded systems. Tiny ML allows for on-device inference without relying on connectivity to the cloud, edge, or even the processing power of mobile devices. This approach is crucial for applications where size, power consumption, and cost are critical factors. Tiny ML devices typically operate with less than 1 MB of RAM and flash memory, consuming only milliwatts of power, enabling battery life of months or years. Applications include wake word detection, gesture recognition, and predictive maintenance in industrial settings. Platforms like Arduino Nano 33 BLE Sense and STM32 microcontrollers, coupled with frameworks like TensorFlow Lite for Microcontrollers, enable ML on these tiny devices. However, Tiny ML requires significant model optimization and quantization[0] to fit within these constraints.

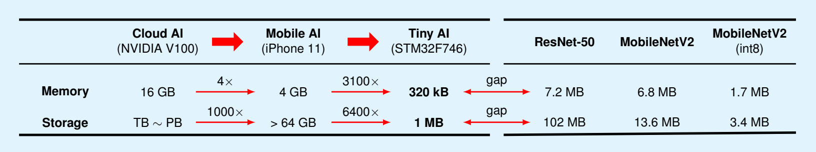
Each of these paradigms has its own strengths and is suited to different use cases:

- Cloud ML remains essential for tasks requiring massive computational power or large-scale data analysis.
- Edge ML is ideal for applications needing low-latency responses or local data processing in industrial or enterprise environments.

[0] Quantization: Process of reducing the numerical precision of ML model parameters to reduce memory footprint and computational demand.

- Mobile ML is suited for personalized, responsive applications on smartphones and tablets.
- Tiny ML enables AI capabilities in small, power-efficient devices, expanding the reach of ML to new domains.

This progression reflects a broader trend in computing towards more distributed, localized, and specialized processing. The evolution is driven by the need for faster response times, improved privacy, reduced bandwidth usage, and the ability to operate in environments with limited or no connectivity, while also catering to the specific capabilities and constraints of different types of devices.



	Cloud AI (NVIDIA V100)	➡	Mobile AI (iPhone 11)	➡	Tiny AI (STM32F746)		ResNet-50	MobileNetV2	MobileNetV2 (int8)
Memory	16 GB	4×	4 GB	3100×	**320 kB**	gap	7.2 MB	6.8 MB	1.7 MB
Storage	TB ~ PB	1000×	> 64 GB	6400×	**1 MB**	gap	102 MB	13.6 MB	3.4 MB

Figure 2.3: From cloud GPUs to microcontrollers: Navigating the memory and storage landscape across computing devices. Source: (Ji Lin, Zhu, et al. 2023)

Figure 2.3 illustrates the key differences between Cloud ML, Edge ML, Mobile ML, and Tiny ML in terms of hardware, latency, connectivity, power requirements, and model complexity. As we move from Cloud to Edge to Tiny ML, we see a dramatic reduction in available resources, which presents significant challenges for deploying sophisticated machine learning models. This resource disparity becomes particularly apparent when attempting to deploy deep learning models on microcontrollers, the primary hardware platform for Tiny ML. These tiny devices have severely constrained memory and storage capacities, which are often insufficient for conventional deep learning models. We will learn to put these things into perspective in this chapter.

2.2 Cloud-Based Machine Learning

The vast computational demands of modern machine learning often require the scalability and power of centralized cloud[1] infrastructures. Cloud Machine Learning (Cloud ML) handles tasks such as large-scale data processing, collaborative model development, and advanced analytics. Cloud data centers leverage distributed architectures, offering specialized resources to train complex models and support diverse applications, from recommendation systems[2] to natural language processing[3].

> ℹ Definition of Cloud ML
>
> **Cloud Machine Learning (Cloud ML)** refers to the deployment of machine learning models on *centralized computing infrastructures*, such as data centers. These systems operate in the *kilowatt to megawatt* power range and utilize *specialized computing systems* to handle *large-scale datasets* and train *complex models*. Cloud ML offers *scalability* and *computational capacity*, making it well-suited for tasks requiring extensive resources

[1] The cloud refers to networks of remote computing servers that provide scalable storage, processing power, and specialized services for deploying machine learning models.

[2] Recommendation systems: An AI technology used to personalize user experiences by predicting and showcasing what users would enjoy or find suitable based on their past behavior or interactions.

[3] Natural Language Processing (NLP): A branch of AI that gives machines the ability to read, understand and derive meaning from human languages to perform tasks like translation, sentiment analysis, and topic classification.

> and collaboration. However, it depends on *consistent connectivity* and may introduce *latency* for real-time applications.

Figure 2.4 provides an overview of Cloud ML's capabilities, which we will discuss in greater detail throughout this section.

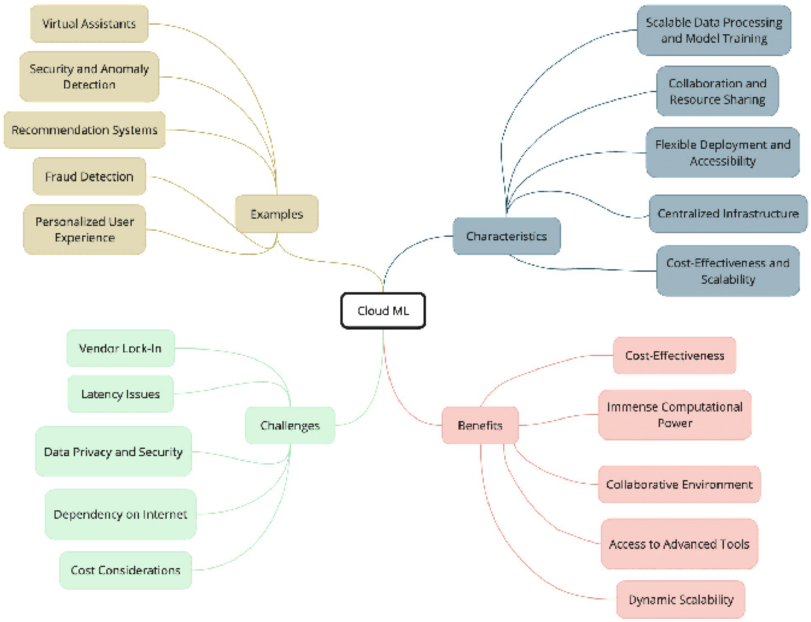

Virtual Assistants
Security and Anomaly Detection
Recommendation Systems
Fraud Detection
Personalized User Experience
Examples
Scalable Data Processing and Model Training
Collaboration and Resource Sharing
Flexible Deployment and Accessibility
Centralized Infrastructure
Cost-Effectiveness and Scalability
Characteristics
Cloud ML
Vendor Lock-In
Latency Issues
Data Privacy and Security
Dependency on Internet
Cost Considerations
Challenges
Benefits
Cost-Effectiveness
Immense Computational Power
Collaborative Environment
Access to Advanced Tools
Dynamic Scalability

Figure 2.4: Section overview for Cloud ML.

2.2.1 Characteristics

One of the key characteristics of Cloud ML is its centralized infrastructure. Figure 2.5 illustrates this concept with an example from Google's Cloud TPU[4] data center. Cloud service providers offer a **virtual platform**[5] that consists of high-capacity servers, expansive storage solutions, and robust networking architectures, all housed in data centers distributed across the globe. As shown in the figure, these centralized facilities can be massive in scale, housing rows upon rows of specialized hardware. This centralized setup allows for the pooling and efficient management of computational resources, making it easier to scale machine learning projects as needed.[6]

Cloud ML excels in its ability to process and analyze massive volumes of data. The centralized infrastructure is designed to handle complex computations and model training tasks that require significant computational power. By leveraging the scalability of the cloud, machine learning models can be trained on vast amounts of data, leading to improved learning capabilities and predictive performance.

[4] Tensor Processing Units (TPUs) are Google's custom-designed AI accelerator chips optimized for machine learning workloads, particularly deep neural network training and inference.

[5] Virtual platforms abstract physical hardware through software interfaces, enabling efficient resource management and automated scaling across multiple users without direct hardware interaction.

[6] While centralized infrastructure enables efficient resource management and scalability, increasing physical distance between data centers and end-users can introduce latency and data privacy challenges.

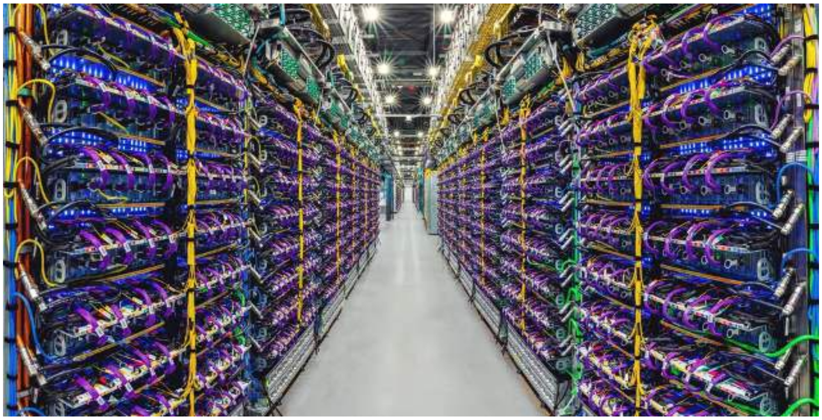


Figure 2.5: Cloud TPU data center at Google. Source: Google.

Another advantage of Cloud ML is the flexibility it offers in terms of deployment and accessibility. Once a machine learning model is trained and validated, it can be deployed through cloud-based APIs and services, making it accessible to users worldwide. This enables seamless integration of ML capabilities into applications across mobile, web, and IoT platforms[7], regardless of the end user's computational resources.

Cloud ML promotes collaboration and resource sharing among teams and organizations. The centralized nature of the cloud infrastructure enables multiple data scientists and engineers to access and work on the same machine learning projects simultaneously. This collaborative approach facilitates knowledge sharing, accelerates the development cycle from experimentation to production, and optimizes resource utilization across teams.

By leveraging the pay-as-you-go pricing model offered by cloud service providers, Cloud ML allows organizations to avoid the upfront capital expenditure[8] associated with building and maintaining dedicated ML infrastructure. The ability to scale resources up during intensive training periods and down during lower demand ensures cost-effectiveness and financial flexibility in managing machine learning projects.

Cloud ML has revolutionized the way machine learning is approached, democratizing access to advanced AI capabilities and making them more accessible, scalable, and efficient. It has enabled organizations of all sizes to harness the power of machine learning without requiring specialized hardware expertise or significant infrastructure investments.

2.2.2 Benefits

Cloud ML offers several significant benefits that make it a powerful choice for machine learning projects:

One of the key advantages of Cloud ML is its ability to provide vast computational resources. The cloud infrastructure is designed to handle complex algorithms and process large datasets efficiently. This is particularly beneficial for machine learning models that require significant computational power, such

[7] Internet of Things (IoT): A system of interrelated computing devices, mechanical and digital machines, capable of transferring data over a network without human-to-human or human-to-computer interaction.

[8] Capital Expenditure (CapEx): Funds used by a company to acquire or upgrade physical assets such as property, industrial buildings or equipment.

as deep learning networks or models trained on massive datasets. By leveraging the cloud's computational capabilities, organizations can overcome the limitations of local hardware setups and scale their machine learning projects to meet demanding requirements.

Cloud ML offers dynamic scalability, allowing organizations to easily adapt to changing computational needs. As the volume of data grows or the complexity of machine learning models increases, the cloud infrastructure can seamlessly scale up or down to accommodate these changes. This flexibility ensures consistent performance and enables organizations to handle varying workloads without the need for extensive hardware investments. With Cloud ML, resources can be allocated on-demand, providing a cost-effective and efficient solution for managing machine learning projects.

Cloud ML platforms provide access to a wide range of advanced tools and algorithms specifically designed for machine learning. These tools often include pre-built models, AutoML capabilities, and specialized APIs that simplify the development and deployment of machine learning solutions. Developers can leverage these resources to accelerate the building, training, and optimization of sophisticated models. By utilizing the latest advancements in machine learning algorithms and techniques, organizations can implement state-of-the-art solutions without needing to develop them from scratch.

Cloud ML fosters a collaborative environment that enables teams to work together seamlessly. The centralized nature of the cloud infrastructure allows multiple data scientists and engineers to access and contribute to the same machine learning projects simultaneously. This collaborative approach facilitates knowledge sharing, promotes cross-functional collaboration, and accelerates the development and iteration of machine learning models. Teams can easily share code, datasets, and results through version control and project management tools integrated with cloud platforms.

Adopting Cloud ML can be a cost-effective solution for organizations, especially compared to building and maintaining an on-premises machine learning infrastructure. Cloud service providers offer flexible pricing models, such as pay-as-you-go or subscription-based plans, allowing organizations to pay only for the resources they consume. This eliminates the need for upfront capital investments in specialized hardware like GPUs and TPUs, reducing the overall cost of implementing machine learning projects. Additionally, the ability to automatically scale down resources during periods of low utilization ensures organizations only pay for what they actually use.

The benefits of Cloud ML, including its immense computational power, dynamic scalability, access to advanced tools and algorithms, collaborative environment, and cost-effectiveness, make it a compelling choice for organizations looking to harness the potential of machine learning. By leveraging the capabilities of the cloud, organizations can accelerate their machine learning initiatives, drive innovation, and gain a competitive edge in today's data-driven landscape.

2.2.3 Challenges

While Cloud ML offers numerous benefits, it also comes with certain challenges that organizations need to consider:

Latency is a primary concern in Cloud ML, particularly for applications requiring real-time responses. The process of transmitting data to centralized cloud servers for processing and then back to applications introduces delays. This can significantly impact time-sensitive scenarios like autonomous vehicles, real-time fraud detection, and industrial control systems where immediate decision-making is crucial. Organizations must implement careful system design to minimize latency and ensure acceptable response times.

Data privacy and security represent critical challenges when centralizing processing and storage in the cloud. Sensitive data transmitted to remote data centers becomes potentially vulnerable to cyber-attacks and unauthorized access. Cloud environments often attract hackers seeking to exploit vulnerabilities in valuable information repositories. Organizations must implement robust security measures including encryption, strict access controls, and continuous monitoring. Additionally, compliance with regulations like GDPR or HIPAA[9] becomes increasingly complex when handling sensitive data in cloud environments.

Cost management becomes increasingly important as data processing requirements grow. Although Cloud ML provides scalability and flexibility, organizations processing large data volumes may experience escalating costs with increased cloud resource consumption. The pay-as-you-go pricing model can quickly accumulate expenses, especially for compute-intensive operations like model training and inference. Effective cloud adoption requires careful monitoring and optimization of usage patterns. Organizations should consider implementing data compression techniques, efficient algorithmic design, and resource allocation optimization to balance cost-effectiveness with performance requirements.

Network dependency presents another significant challenge for Cloud ML implementations. The requirement for stable and reliable internet connectivity means that any disruptions in network availability directly impact system performance. This dependency becomes particularly problematic in environments with limited, unreliable, or expensive network access. Building resilient ML systems requires robust network infrastructure complemented by appropriate failover mechanisms or offline processing capabilities.

Vendor lock-in often emerges as organizations adopt specific tools, APIs, and services from their chosen cloud provider. This dependency can complicate future transitions between providers or platform migrations. Organizations may encounter challenges with portability, interoperability, and cost implications when considering changes to their cloud ML infrastructure. Strategic planning should include careful evaluation of vendor offerings, consideration of long-term goals, and preparation for potential migration scenarios to mitigate lock-in risks.

Addressing these challenges requires thorough planning, thoughtful architectural design, and comprehensive risk mitigation strategies. Organizations must balance Cloud ML benefits against potential challenges based on their specific requirements, data sensitivity concerns, and business objectives. Proactive approaches to these challenges enable organizations to effectively leverage Cloud ML while maintaining data privacy, security, cost-effectiveness, and system reliability.

[9] GDPR (General Data Protection Regulation) and HIPAA (Health Insurance Portability and Accountability Act): Regulations governing data protection and maintaining data privacy in EU and US respectively.

2.2.4 Use Cases

Cloud ML has found widespread adoption across various domains, revolutionizing the way businesses operate and users interact with technology. Let's explore some notable examples of Cloud ML in action:

Cloud ML plays a crucial role in powering virtual assistants like Siri and Alexa. These systems leverage the immense computational capabilities of the cloud to process and analyze voice inputs in real-time. By harnessing the power of natural language processing and machine learning algorithms, virtual assistants can understand user queries, extract relevant information, and generate intelligent and personalized responses. The cloud's scalability and processing power enable these assistants to handle a vast number of user interactions simultaneously, providing a seamless and responsive user experience.[10]

Cloud ML forms the backbone of advanced recommendation systems used by platforms like Netflix and Amazon. These systems use the cloud's ability to process and analyze massive datasets to uncover patterns, preferences, and user behavior. By leveraging collaborative filtering and other machine learning techniques, recommendation systems can offer personalized content or product suggestions tailored to each user's interests. The cloud's scalability allows these systems to continuously update and refine their recommendations based on the ever-growing amount of user data, enhancing user engagement and satisfaction.

In the financial industry, Cloud ML has revolutionized fraud detection systems. By leveraging the cloud's computational power, these systems can analyze vast amounts of transactional data in real-time to identify potential fraudulent activities. Machine learning algorithms trained on historical fraud patterns can detect anomalies and suspicious behavior, enabling financial institutions to take proactive measures to prevent fraud and minimize financial losses. The cloud's ability to process and store large volumes of data makes it an ideal platform for implementing robust and scalable fraud detection systems.

Cloud ML is deeply integrated into our online experiences, shaping the way we interact with digital platforms. From personalized ads on social media feeds to predictive text features in email services, Cloud ML powers smart algorithms that enhance user engagement and convenience. It enables e-commerce sites to recommend products based on a user's browsing and purchase history, fine-tunes search engines to deliver accurate and relevant results, and automates the tagging and categorization of photos on platforms like Facebook. By leveraging the cloud's computational resources, these systems can continuously learn and adapt to user preferences, providing a more intuitive and personalized user experience.

Cloud ML plays a role in bolstering user security by powering anomaly detection systems[11]. These systems continuously monitor user activities and system logs to identify unusual patterns or suspicious behavior. By analyzing vast amounts of data in real-time, Cloud ML algorithms can detect potential cyber threats, such as unauthorized access attempts, malware infections, or data breaches. The cloud's scalability and processing power enable these systems to handle the increasing complexity and volume of security data, providing a proactive approach to protecting users and systems from potential threats.

[10] Virtual assistants exemplify hybrid ML architecture by combining local wake word detection via Tiny ML with cloud-based natural language processing. This design optimizes for both power efficiency and sophisticated language understanding capabilities while maintaining responsiveness.

[11] Anomaly Detection Systems: Machine learning systems designed to identify unusual patterns or outliers in the data which may indicate suspicious or abnormal behavior.

2.3 Edge Machine Learning

As machine learning applications grow, so does the need for faster, localized decision-making. Edge Machine Learning (Edge ML) shifts computation away from centralized servers, processing data closer to its source. This paradigm is critical for time-sensitive applications, such as autonomous systems, industrial IoT, and smart infrastructure, where minimizing latency and preserving data privacy are paramount. Edge devices, like gateways[12] and IoT hubs,[13] enable these systems to function efficiently while reducing dependence on cloud infrastructures.

> **Definition of Edge ML**
>
> **Edge Machine Learning (Edge ML)** describes the deployment of machine learning models at or near the *edge of the network*. These systems operate in the *tens to hundreds of watts* range and rely on *localized hardware* optimized for *real-time processing*. Edge ML minimizes *latency* and enhances *privacy* by processing data locally, but its primary limitation lies in *restricted computational resources*.

Figure 2.6 provides an overview of this section.

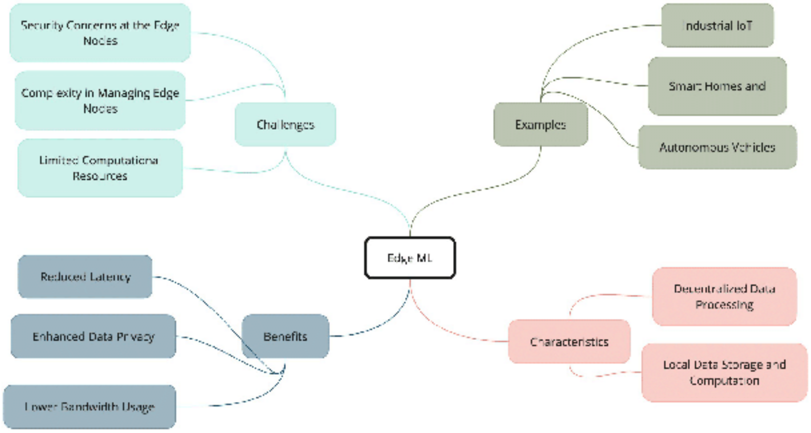

Security Concerns at the Edge Nodes
Complexity in Managing Edge Nodes
Limited Computational Resources
Challenges
Examples
Industrial IoT
Smart Homes and
Autonomous Vehicles
Edge ML
Reduced Latency
Enhanced Data Privacy
Lower Bandwidth Usage
Benefits
Characteristics
Decentralized Data Processing
Local Data Storage and Computation

Figure 2.6: Section overview for Edge ML.

2.3.1 Characteristics

In Edge ML, data processing happens in a decentralized fashion, as illustrated in Figure 2.7. Instead of sending data to remote servers, the data is processed locally on devices like smartphones, tablets, or Internet of Things (IoT) devices. The figure showcases various examples of these edge devices, including wearables, industrial sensors, and smart home appliances. This local processing allows devices to make quick decisions based on the data they collect without relying heavily on a central server's resources.

[12] Gateways: Network nodes that act as a bridge between different networks.

[13] IoT Hubs: Devices or services that manage data communication between IoT devices and the cloud.

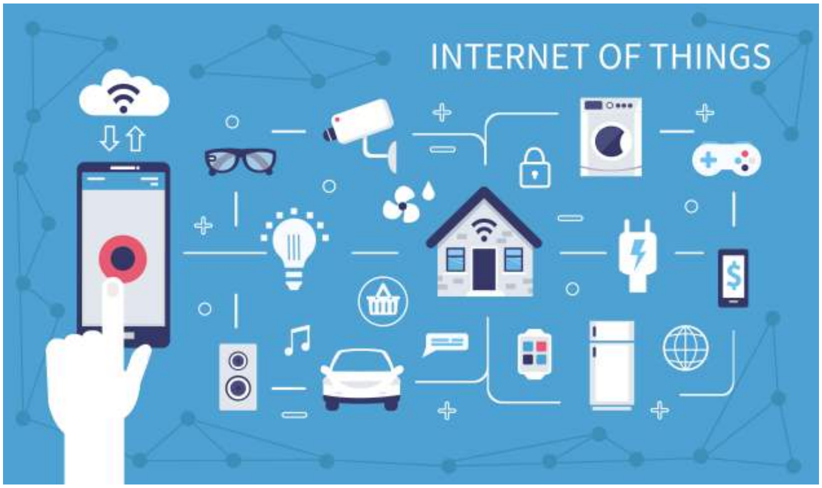


Figure 2.7: Edge ML Examples. Source: Edge Impulse.

Local data storage and computation are key features of Edge ML. This setup ensures that data can be stored and analyzed directly on the devices, thereby maintaining the privacy of the data and reducing the need for constant internet connectivity. Moreover, this approach reduces latency in decision-making processes, as computations occur closer to where data is generated. This proximity not only enhances real-time capabilities but also often results in more efficient resource utilization, as data doesn't need to travel across networks, saving bandwidth and energy consumption.

2.3.2 Benefits

One of Edge ML's main advantages is the significant latency reduction compared to Cloud ML. This reduced latency can be a critical benefit in situations where milliseconds count, such as in autonomous vehicles, where quick decision-making can mean the difference between safety and an accident.

Edge ML also offers improved data privacy, as data is primarily stored and processed locally. This minimizes the risk of data breaches that are more common in centralized data storage solutions. Sensitive information can be kept more secure, as it's not sent over networks that could be intercepted.

Operating closer to the data source means less data must be sent over networks, reducing bandwidth usage. This can result in cost savings and efficiency gains, especially in environments where bandwidth is limited or costly.

2.3.3 Challenges

However, Edge ML has its challenges. One of the main concerns is the limited computational resources compared to cloud-based solutions. Endpoint devices may have a different processing power or storage capacity than cloud servers, limiting the complexity of the machine learning models that can be deployed.

Managing a network of edge nodes can introduce complexity, especially regarding coordination, updates, and maintenance. Ensuring all nodes operate seamlessly and are up-to-date with the latest algorithms and security protocols can be a logistical challenge.

While Edge ML offers enhanced data privacy, edge nodes can sometimes be more vulnerable to physical and cyber-attacks. Developing robust security protocols that protect data at each node without compromising the system's efficiency remains a significant challenge in deploying Edge ML solutions.

2.3.4 Use Cases

Edge ML has many applications, from autonomous vehicles and smart homes to industrial Internet of Things (IoT). These examples were chosen to highlight scenarios where real-time data processing, reduced latency, and enhanced privacy are not just beneficial but often critical to the operation and success of these technologies. They demonstrate the role that Edge ML can play in driving advancements in various sectors, fostering innovation, and paving the way for more intelligent, responsive, and adaptive systems.

Autonomous vehicles stand as a prime example of Edge ML's potential. These vehicles rely heavily on real-time data processing to navigate and make decisions. Localized machine learning models assist in quickly analyzing data from various sensors to make immediate driving decisions, ensuring safety and smooth operation.

Edge ML plays a crucial role in efficiently managing various systems in smart homes and buildings, from lighting and heating to security. By processing data locally, these systems can operate more responsively and harmoniously with the occupants' habits and preferences, creating a more comfortable living environment.

The Industrial IoT[14] leverages Edge ML to monitor and control complex industrial processes. Here, machine learning models can analyze data from numerous sensors in real-time, enabling predictive maintenance, optimizing operations, and enhancing safety measures. This revolution in industrial automation and efficiency is transforming manufacturing and production across various sectors.

The applicability of Edge ML is vast and not limited to these examples. Various other sectors, including healthcare, agriculture, and urban planning, are exploring and integrating Edge ML to develop innovative solutions responsive to real-world needs and challenges, heralding a new era of smart, interconnected systems.

2.4 Mobile Machine Learning

Machine learning is increasingly being integrated into portable devices like smartphones and tablets, empowering users with real-time, personalized capabilities. Mobile Machine Learning (Mobile ML) supports applications like voice recognition, computational photography, and health monitoring, all while maintaining data privacy through on-device computation. These battery-powered devices are optimized for responsiveness and can operate offline, making them indispensable in everyday consumer technologies.

[14] Industrial IoT (IoT) encompasses interconnected sensors, instruments, and devices networked together within industrial applications. It enables data collection, exchange, and analysis to improve manufacturing and industrial processes through machine learning and automation.

ℹ Definition of Mobile ML

Mobile Machine Learning (Mobile ML) enables machine learning models to run directly on *portable, battery-powered devices* like smartphones and tablets. Operating within the *single-digit to tens of watts* range, Mobile ML leverages *on-device computation* to provide *personalized and responsive applications*. This paradigm preserves *privacy* and ensures *offline functionality*, though it must balance *performance* with *battery and storage limitations*.

2.4.1 Characteristics

Mobile ML utilizes the processing power of mobile devices' System-on-Chip (SoC)[15] architectures, including specialized Neural Processing Units (NPUs)[16] and AI accelerators. This enables efficient execution of ML models directly on the device, allowing for real-time processing of data from device sensors like cameras, microphones, and motion sensors without constant cloud connectivity.

Mobile ML is supported by specialized frameworks and tools designed specifically for mobile deployment, such as TensorFlow Lite for Android devices and Core ML for iOS devices. These frameworks are optimized for mobile hardware and provide efficient model compression[17] and quantization techniques to ensure smooth performance within mobile resource constraints.

2.4.2 Benefits

Mobile ML enables real-time processing of data directly on mobile devices, eliminating the need for constant server communication. This results in faster response times for applications requiring immediate feedback, such as real-time translation, face detection, or gesture recognition.

By processing data locally on the device, Mobile ML helps maintain user privacy. Sensitive information doesn't need to leave the device, reducing the risk of data breaches and addressing privacy concerns, particularly important for applications handling personal data.

Mobile ML applications can function without constant internet connectivity, making them reliable in areas with poor network coverage or when users are offline. This ensures consistent performance and user experience regardless of network conditions.

2.4.3 Challenges

Despite modern mobile devices being powerful, they still face resource constraints compared to cloud servers. Mobile ML must operate within limited RAM, storage, and processing power, requiring careful optimization of models and efficient resource management.

ML operations can be computationally intensive, potentially impacting device battery life. Developers must balance model complexity and performance with power consumption to ensure reasonable battery life for users.

Mobile devices have limited storage space, necessitating careful consideration of model size. This often requires model compression and quantization techniques, which can affect model accuracy and performance.

[15] System-on-Chip (SoC): An integrated circuit that packages essential components of a computer or other system into a single chip.

[16] Neural Processing Unit (NPU): A specialized hardware unit designed for accelerated processing of AI and machine learning algorithms.

[17] Model compression reduces ML model size through techniques like pruning, quantization, and knowledge distillation. This process decreases memory requirements and computational demands while preserving key model functionality, enabling efficient deployment on resource-constrained devices.

2.4.4 Use Cases

Mobile ML has revolutionized how we use cameras on mobile devices, enabling sophisticated computer vision applications that process visual data in real-time. Modern smartphone cameras now incorporate ML models that can detect faces, analyze scenes, and apply complex filters instantaneously. These models work directly on the camera feed to enable features like portrait mode photography, where ML algorithms separate foreground subjects from backgrounds. Document scanning applications use ML to detect paper edges, correct perspective, and enhance text readability, while augmented reality applications use ML-powered object detection to accurately place virtual objects in the real world.

Natural language processing on mobile devices has transformed how we interact with our phones and communicate with others. Speech recognition models run directly on device, enabling voice assistants to respond quickly to commands even without internet connectivity. Real-time translation applications can now translate conversations and text without sending data to the cloud, preserving privacy and working reliably regardless of network conditions. Mobile keyboards have become increasingly intelligent, using ML to predict not just the next word but entire phrases based on the user's writing style and context, while maintaining all learning and personalization locally on the device.

Mobile ML has enabled smartphones and tablets to become sophisticated health monitoring devices. Through clever use of existing sensors combined with ML models, mobile devices can now track physical activity, analyze sleep patterns, and monitor vital signs. For example, cameras can measure heart rate by detecting subtle color changes in the user's skin, while accelerometers and ML models work together to recognize specific exercises and analyze workout form. These applications process sensitive health data directly on the device, ensuring privacy while providing users with real-time feedback and personalized health insights.

Perhaps the most pervasive but least visible application of Mobile ML lies in how it personalizes and enhances the overall user experience. ML models continuously analyze how users interact with their devices to optimize everything from battery usage to interface layouts. These models learn individual usage patterns to predict which apps users are likely to open next, preload content they might want to see, and adjust system settings like screen brightness and audio levels based on environmental conditions and user preferences. This creates a deeply personalized experience that adapts to each user's needs while maintaining privacy by keeping all learning and adaptation on the device itself.

These applications demonstrate how Mobile ML bridges the gap between cloud-based solutions and edge computing, providing efficient, privacy-conscious, and user-friendly machine learning capabilities on personal mobile devices. The continuous advancement in mobile hardware capabilities and optimization techniques continues to expand the possibilities for Mobile ML applications.

2.5 Tiny Machine Learning

Tiny Machine Learning (Tiny ML) brings intelligence to the smallest devices, from microcontrollers[18] to embedded sensors, enabling real-time computation in resource-constrained environments. These systems power applications such as predictive maintenance, environmental monitoring, and simple gesture recognition. Tiny ML devices are optimized for energy efficiency, often running for months or years on limited power sources, such as coin-cell batteries, while delivering actionable insights in remote or disconnected environments.

> Definition of Tiny ML
>
> **Tiny Machine Learning (Tiny ML)** refers to the execution of machine learning models on *ultra-constrained devices*, such as microcontrollers and sensors. These devices operate in the *milliwatt to sub-watt* power range, prioritizing *energy efficiency* and *compactness*. Tiny ML enables *localized decision-making* in resource-constrained environments, excelling in applications where *extended operation on limited power sources* is required. However, it is limited by *severely restricted computational resources*.

Figure 2.8 encapsulates the key aspects of Tiny ML discussed in this section.

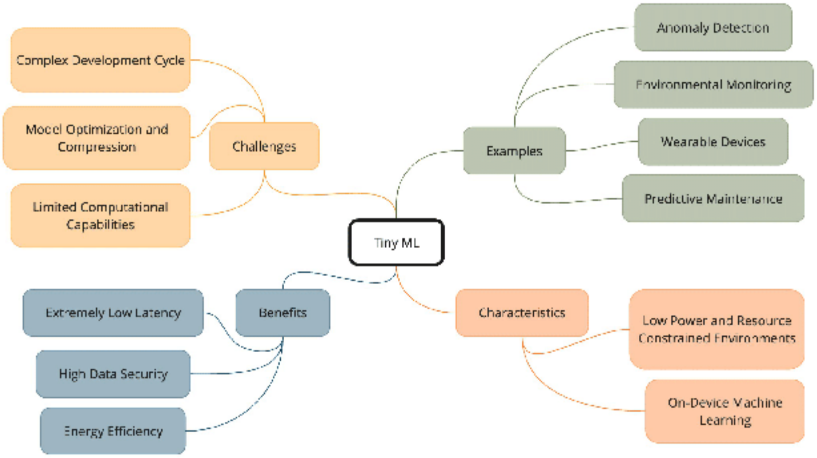


Figure 2.8: Section overview for Tiny ML.

2.5.1 Characteristics

In Tiny ML, the focus, much like in Mobile ML, is on on-device machine learning. This means that machine learning models are deployed and trained on the device, eliminating the need for external servers or cloud infrastructures. This allows Tiny ML to enable intelligent decision-making right where the data is generated, making real-time insights and actions possible, even in settings where connectivity is limited or unavailable.

[18] Microcontroller: A compact, low-cost computing device designed for control-oriented applications. Includes an integrated CPU, memory, and peripherals.

Tiny ML excels in low-power and resource-constrained settings. These environments require highly optimized solutions that function within the available resources. Figure 2.9 showcases an example Tiny ML device kit, illustrating the compact nature of these systems. These devices can typically fit in the palm of your hand or, in some cases, are even as small as a fingernail. Tiny ML meets the need for efficiency through specialized algorithms and models designed to deliver decent performance while consuming minimal energy, thus ensuring extended operational periods, even in battery-powered devices like those shown.

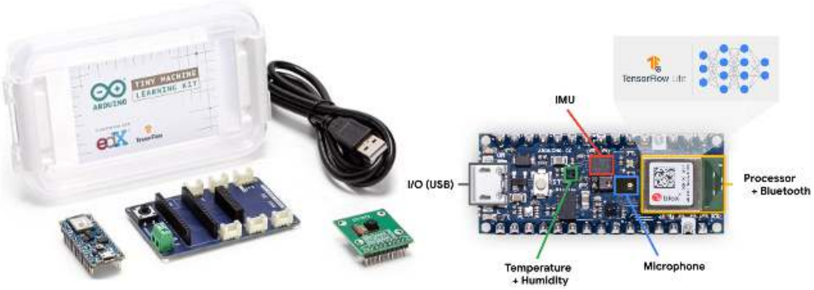


Figure 2.9: Examples of Tiny ML device kits. Source: Widening Access to Applied Machine Learning with Tiny ML.

2.5.2 Benefits

One of the standout benefits of Tiny ML is its ability to offer ultra-low latency. Since computation occurs directly on the device, the time required to send data to external servers and receive a response is eliminated. This is crucial in applications requiring immediate decision-making, enabling quick responses to changing conditions.

Tiny ML inherently enhances data security. Because data processing and analysis happen on the device, the risk of data interception during transmission is virtually eliminated. This localized approach to data management ensures that sensitive information stays on the device, strengthening user data security.

Tiny ML operates within an energy-efficient framework, a necessity given its resource-constrained environments. By employing lean algorithms and optimized computational methods, Tiny ML ensures that devices can execute complex tasks without rapidly depleting battery life, making it a sustainable option for long-term deployments.

2.5.3 Challenges

However, the shift to Tiny ML comes with its set of hurdles. The primary limitation is the devices' constrained computational capabilities. The need to operate within such limits means that deployed models must be simplified, which could affect the accuracy and sophistication of the solutions.

Tiny ML also introduces a complicated development cycle. Crafting lightweight and effective models demands a deep understanding of machine learning principles and expertise in embedded systems. This complexity calls for a collaborative development approach, where multi-domain expertise is essential for success.

A central challenge in Tiny ML is model optimization and compression. Creating machine learning models that can operate effectively within the limited memory and computational power of microcontrollers requires innovative approaches to model design. Developers often face the challenge of striking a delicate balance and optimizing models to maintain effectiveness while fitting within stringent resource constraints.

2.5.4 Use Cases

In wearables, Tiny ML opens the door to smarter, more responsive gadgets. From fitness trackers offering real-time workout feedback to smart glasses processing visual data on the fly, Tiny ML transforms how we engage with wearable tech, delivering personalized experiences directly from the device.

In industrial settings, Tiny ML plays a significant role in predictive maintenance[19]. By deploying Tiny ML algorithms on sensors that monitor equipment health, companies can preemptively identify potential issues, reducing downtime and preventing costly breakdowns. On-site data analysis ensures quick responses, potentially stopping minor issues from becoming major problems.

Tiny ML can be employed to create anomaly detection models that identify unusual data patterns. For instance, a smart factory could use Tiny ML to monitor industrial processes and spot anomalies, helping prevent accidents and improve product quality. Similarly, a security company could use Tiny ML to monitor network traffic for unusual patterns, aiding in detecting and preventing cyber-attacks. Tiny ML could monitor patient data for anomalies in healthcare, aiding early disease detection and better patient treatment.

In environmental monitoring, Tiny ML enables real-time data analysis from various field-deployed sensors. These could range from city air quality monitoring to wildlife tracking in protected areas. Through Tiny ML, data can be processed locally, allowing for quick responses to changing conditions and providing a nuanced understanding of environmental patterns, crucial for informed decision-making.

In summary, Tiny ML serves as a trailblazer in the evolution of machine learning, fostering innovation across various fields by bringing intelligence directly to the edge. Its potential to transform our interaction with technology and the world is immense, promising a future where devices are connected, intelligent, and capable of making real-time decisions and responses.

2.6 Hybrid Machine Learning

The increasingly complex demands of modern applications often require a blend of machine learning approaches. Hybrid Machine Learning (Hybrid ML) combines the computational power of the cloud, the efficiency of edge and mobile devices, and the compact capabilities of Tiny ML. This approach enables architects to create systems that balance performance, privacy, and resource efficiency, addressing real-world challenges with innovative, distributed solutions.

[19] Predictive maintenance refers to the use of data-driven, proactive maintenance methods that predict equipment failures.

ℹ Definition of Hybrid ML

Hybrid Machine Learning (Hybrid ML) refers to the integration of multiple ML paradigms, such as Cloud, Edge, Mobile, and Tiny ML, to form a unified, distributed system. These systems leverage the *complementary strengths* of each paradigm while addressing their *individual limitations*. Hybrid ML supports *scalability, adaptability,* and *privacy-preserving capabilities,* enabling sophisticated ML applications for diverse scenarios. By combining centralized and decentralized computing, Hybrid ML facilitates efficient resource utilization while meeting the demands of complex real-world requirements.

2.6.1 Design Patterns

Design patterns in Hybrid ML represent reusable solutions to common challenges faced when integrating multiple ML paradigms (cloud, edge, mobile, and tiny). These patterns guide system architects in combining the strengths of different approaches, including the computational power of the cloud and the efficiency of edge devices, while mitigating their individual limitations. By following these patterns, architects can address key trade-offs in performance, latency, privacy, and resource efficiency.

Hybrid ML design patterns serve as blueprints, enabling the creation of scalable, efficient, and adaptive systems tailored to diverse real-world applications. Each pattern reflects a specific strategy for organizing and deploying ML workloads across different tiers of a distributed system, ensuring optimal use of available resources while meeting application-specific requirements.

2.6.1.1 Train-Serve Split

One of the most common hybrid patterns is the train-serve split, where model training occurs in the cloud but inference happens on edge, mobile, or tiny devices. This pattern takes advantage of the cloud's vast computational resources for the training phase while benefiting from the low latency and privacy advantages of on-device inference. For example, smart home devices often use models trained on large datasets in the cloud but run inference locally to ensure quick response times and protect user privacy. In practice, this might involve training models on powerful systems like the NVIDIA DGX A100, leveraging its 8 A100 GPUs and terabyte-scale memory, before deploying optimized versions to edge devices like the NVIDIA Jetson AGX Orin for efficient inference. Similarly, mobile vision models for computational photography are typically trained on powerful cloud infrastructure but deployed to run efficiently on phone hardware.

2.6.1.2 Hierarchical Processing

Hierarchical processing creates a multi-tier system where data and intelligence flow between different levels of the ML stack. In industrial IoT applications, tiny sensors might perform basic anomaly detection, edge devices aggregate and analyze data from multiple sensors, and cloud systems handle complex analytics and model updates. For instance, we might see ESP32-CAM devices performing basic image classification at the sensor level with their minimal 520 KB RAM, feeding data up to Jetson AGX Orin devices for more sophisticated computer vision tasks, and ultimately connecting to cloud infrastructure for complex analytics and model updates.

This hierarchy allows each tier to handle tasks appropriate to its capabilities. Tiny ML devices handle immediate, simple decisions; edge devices manage local coordination; and cloud systems tackle complex analytics and learning tasks. Smart city installations often use this pattern, with street-level sensors feeding data to neighborhood-level edge processors, which in turn connect to city-wide cloud analytics.

2.6.1.3 Progressive Deployment

Progressive deployment strategies adapt models for different computational tiers, creating a cascade of increasingly lightweight versions. A model might start as a large, complex version in the cloud, then be progressively compressed and optimized for edge servers, mobile devices, and finally tiny sensors. Voice assistant systems often employ this pattern, where full natural language processing runs in the cloud, while simplified wake-word detection[20] runs on-device. This allows the system to balance capability and resource constraints across the ML stack.

2.6.1.4 Federated Learning

Federated learning represents a sophisticated hybrid approach where model training is distributed across many edge or mobile devices while maintaining privacy. Devices learn from local data and share model updates, rather than raw data, with cloud servers that aggregate these updates into an improved global model. This pattern is particularly powerful for applications like keyboard prediction on mobile devices or healthcare analytics, where privacy is paramount but benefits from collective learning are valuable. The cloud coordinates the learning process without directly accessing sensitive data, while devices benefit from the collective intelligence of the network.

2.6.1.5 Collaborative Learning

Collaborative learning enables peer-to-peer learning between devices at the same tier, often complementing hierarchical structures. Autonomous vehicle fleets, for example, might share learning about road conditions or traffic patterns directly between vehicles while also communicating with cloud infrastructure. This horizontal collaboration allows systems to share time-sensitive information and learn from each other's experiences without always routing through central servers.

2.6.2 Real-World Integration

Design patterns establish a foundation for organizing and optimizing ML workloads across distributed systems. However, the practical application of these

[20] Wake-word Detection: The task of detecting a specific phrase (wake word) used to activate a voice-controlled system.

patterns often requires combining multiple paradigms into integrated workflows. Thus, in practice, ML systems rarely operate in isolation. Instead, they form interconnected networks where each paradigm, including Cloud, Edge, Mobile, and Tiny ML, plays a specific role while communicating with other parts of the system. These interconnected networks follow integration patterns that assign specific roles to Cloud, Edge, Mobile, and Tiny ML systems based on their unique strengths and limitations. Recall that cloud systems excel at training and analytics but require significant infrastructure. Edge systems provide local processing power and reduced latency. Mobile devices offer personal computing capabilities and user interaction. Tiny ML enables intelligence in the smallest devices and sensors.

Figure 2.10 illustrates these key interactions through specific connection types: "Deploy" paths show how models flow from cloud training to various devices, "Data" and "Results" show information flow from sensors through processing stages, "Analyze" shows how processed information reaches cloud analytics, and "Sync" demonstrates device coordination. Notice how data generally flows upward from sensors through processing layers to cloud analytics, while model deployments flow downward from cloud training to various inference points. The interactions aren't strictly hierarchical. Mobile devices might communicate directly with both cloud services and tiny sensors, while edge systems can assist mobile devices with complex processing tasks.

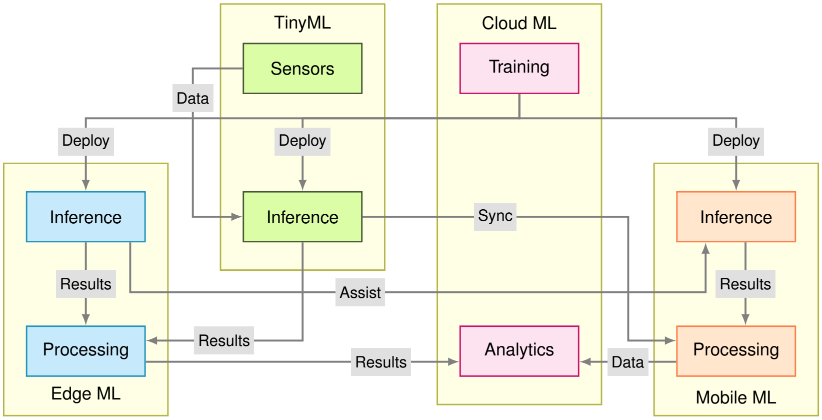


Figure 2.10: Example interaction patterns between ML paradigms, showing data flows, model deployment, and processing relationships across Cloud, Edge, Mobile, and Tiny ML systems.

To understand how these labeled interactions manifest in real applications, let's explore several common scenarios using Figure 2.10:

- **Model Deployment Scenario**: A company develops a computer vision model for defect detection. Following the "Deploy" paths shown in Figure 2.10, the cloud-trained model is distributed to edge servers in factories, quality control tablets on the production floor, and tiny cameras embedded in the production line. This showcases how a single ML solution can be distributed across different computational tiers for optimal performance.

- **Data Flow and Analysis Scenario**: In a smart agriculture system, soil sensors (Tiny ML) collect moisture and nutrient data, following the "Data" path to Tiny ML inference. The "Results" flow to edge processors in local stations, which process this information and use the "Analyze" path to send insights to the cloud for farm-wide analytics, while also sharing results with farmers' mobile apps. This demonstrates the hierarchical flow shown in Figure 2.10 from sensors through processing to cloud analytics.
- **Edge-Mobile Assistance Scenario**: When a mobile app needs to perform complex image processing that exceeds the phone's capabilities, it utilizes the "Assist" connection shown in Figure 2.10. The edge system helps process the heavier computational tasks, sending back results to enhance the mobile app's performance. This shows how different ML tiers can cooperate to handle demanding tasks.
- **Tiny ML-Mobile Integration Scenario**: A fitness tracker uses Tiny ML to continuously monitor activity patterns and vital signs. Using the "Sync" pathway shown in Figure 2.10, it synchronizes this processed data with the user's smartphone, which combines it with other health data before sending consolidated updates via the "Analyze" path to the cloud for long-term health analysis. This illustrates the common pattern of tiny devices using mobile devices as gateways to larger networks.
- **Multi-Layer Processing Scenario**: In a smart retail environment, tiny sensors monitor inventory levels, using "Data" and "Results" paths to send inference results to both edge systems for immediate stock management and mobile devices for staff notifications. Following the "Analyze" path, the edge systems process this data alongside other store metrics, while the cloud analyzes trends across all store locations. This demonstrates how the interactions shown in Figure 2.10 enable ML tiers to work together in a complete solution.

These real-world patterns demonstrate how different ML paradigms naturally complement each other in practice. While each approach has its own strengths, their true power emerges when they work together as an integrated system. By understanding these patterns, system architects can better design solutions that effectively leverage the capabilities of each ML tier while managing their respective constraints.

2.7 Shared Principles

The design and integration patterns illustrate how ML paradigms, such as Cloud, Edge, Mobile, and Tiny, interact to address real-world challenges. While each paradigm is tailored to specific roles, their interactions reveal recurring principles that guide effective system design. These shared principles provide a unifying framework for understanding both individual ML paradigms and their hybrid combinations. As we explore these principles, a deeper system design perspective emerges, showing how different ML implementations, which are optimized for distinct contexts, converge around core concepts. This convergence forms the foundation for systematically understanding ML systems, despite their diversity and breadth.

Figure 2.11 illustrates this convergence, highlighting the relationships that underpin practical system design and implementation. Grasping these principles is invaluable not only for working with individual ML systems but also for developing hybrid solutions that leverage their strengths, mitigate their limitations, and create cohesive, efficient ML workflows.

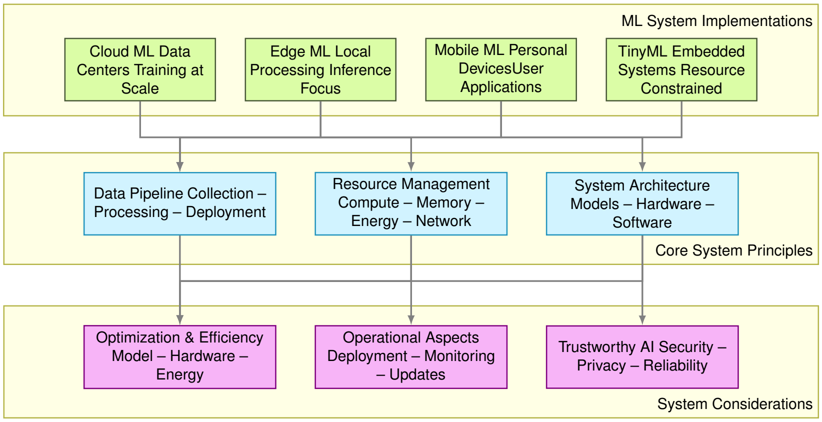

ML System Implementations
Cloud ML Data Centers Training at Scale
Edge ML Local Processing Inference Focus
Mobile ML Personal DevicesUser Applications
TinyML Embedded Systems Resource Constrained
Data Pipeline Collection – Processing – Deployment
Resource Management Compute – Memory – Energy – Network
System Architecture Models – Hardware – Software
Core System Principles
Optimization & Efficiency Model – Hardware – Energy
Operational Aspects Deployment – Monitoring – Updates
Trustworthy AI Security – Privacy – Reliability
System Considerations

Figure 2.11: Core principles converge across different ML system implementations, from cloud to tiny deployments, sharing common foundations in data pipelines, resource management, and system architecture.

The figure shows three key layers that help us understand how ML systems relate to each other. At the top, we see the diverse implementations that we have explored throughout this chapter. Cloud ML operates in data centers, focusing on training at scale with vast computational resources. Edge ML emphasizes local processing with inference capabilities closer to data sources. Mobile ML leverages personal devices for user-centric applications. Tiny ML brings intelligence to highly constrained embedded systems and sensors.

Despite their distinct characteristics, the arrows in the figure show how all these implementations connect to the same core system principles. This reflects an important reality in ML systems, even though they may operate at dramatically different scales, from cloud systems processing petabytes to tiny devices handling kilobytes, they all must solve similar fundamental challenges in terms of:

- Managing data pipelines from collection through processing to deployment
- Balancing resource utilization across compute, memory, energy, and network
- Implementing system architectures that effectively integrate models, hardware, and software

These core principles then lead to shared system considerations around optimization, operations, and trustworthiness. This progression helps explain why techniques developed for one scale of ML system often transfer effectively to others. The underlying problems, efficiently processing data, managing resources, and ensuring reliable operation, remain consistent even as the specific solutions vary based on scale and context.

Understanding this convergence becomes particularly valuable as we move towards hybrid ML systems. When we recognize that different ML implementations share fundamental principles, combining them effectively becomes more intuitive. We can better appreciate why, for example, a cloud-trained model can be effectively deployed to edge devices, or why mobile and tiny ML systems can complement each other in IoT applications.

2.7.1 Implementation Layer

The top layer of Figure 2.11 represents the diverse landscape of ML systems we've explored throughout this chapter. Each implementation addresses specific needs and operational contexts, yet all contribute to the broader ecosystem of ML deployment options.

Cloud ML, centered in data centers, provides the foundation for large-scale training and complex model serving. With access to vast computational resources like the NVIDIA DGX A100 systems we saw in Table 2.1, cloud implementations excel at handling massive datasets and training sophisticated models. This makes them particularly suited for tasks requiring extensive computational power, such as training foundation models[21] or processing large-scale analytics.

Edge ML shifts the focus to local processing, prioritizing inference capabilities closer to data sources. Using devices like the NVIDIA Jetson AGX Orin, edge implementations balance computational power with reduced latency and improved privacy. This approach proves especially valuable in scenarios requiring quick decisions based on local data, such as industrial automation or real-time video analytics.

Mobile ML leverages the capabilities of personal devices, particularly smartphones and tablets. With specialized hardware like Apple's A17 Pro chip, mobile implementations enable sophisticated ML capabilities while maintaining user privacy and providing offline functionality. This paradigm has revolutionized applications from computational photography to on-device speech recognition.

Tiny ML represents the frontier of embedded ML, bringing intelligence to highly constrained devices. Operating on microcontrollers like the Arduino Nano 33 BLE Sense[22], tiny implementations must carefully balance functionality with severe resource constraints. Despite these limitations, Tiny ML enables ML capabilities in scenarios where power efficiency and size constraints are paramount.

2.7.2 System Principles Layer

The middle layer reveals the fundamental principles that unite all ML systems, regardless of their implementation scale. These core principles remain consistent even as their specific manifestations vary dramatically across different deployments.

Data Pipeline principles govern how systems handle information flow, from initial collection through processing to final deployment. In cloud systems, this might mean processing petabytes of data through distributed pipelines. For tiny systems, it could involve carefully managing sensor data streams within

[21] Foundation Models: Large-scale AI models pre-trained on vast amounts of data that can be adapted to a wide range of downstream tasks. Examples include GPT-3, PaLM, and BERT. These models demonstrate emergent capabilities as they scale in size and training data.

[22] The Arduino Nano 33 BLE Sense, introduced in 2019, is a microcontroller specifically designed for Tiny ML applications, featuring sensors and Bluetooth connectivity to facilitate on-device intelligence.

limited memory. Despite these scale differences, all systems must address the same fundamental challenges of data ingestion, transformation, and utilization.

Resource Management emerges as a universal challenge across all implementations. Whether managing thousands of GPUs in a data center or optimizing battery life on a microcontroller, all systems must balance competing demands for computation, memory, energy, and network resources. The quantities involved may differ by orders of magnitude, but the core principles of resource allocation and optimization remain remarkably consistent.

System Architecture principles guide how ML systems integrate models, hardware, and software components. Cloud architectures might focus on distributed computing and scalability, while tiny systems emphasize efficient memory mapping and interrupt handling. Yet all must solve fundamental problems of component integration, data flow optimization, and processing coordination.

2.7.3 System Considerations Layer

The bottom layer of Figure 2.11 illustrates how fundamental principles manifest in practical system-wide considerations. These considerations span all ML implementations, though their specific challenges and solutions vary based on scale and context.

Optimization and Efficiency shape how ML systems balance performance with resource utilization. In cloud environments, this often means optimizing model training across GPU clusters[23] while managing energy consumption in data centers. Edge systems focus on reducing model size and accelerating inference without compromising accuracy. Mobile implementations must balance model performance with battery life and thermal constraints. Tiny ML pushes optimization to its limits, requiring extensive model compression and quantization to fit within severely constrained environments. Despite these different emphases, all implementations grapple with the core challenge of maximizing performance within their available resources.

Operational Aspects affect how ML systems are deployed, monitored, and maintained in production environments. Cloud systems must handle continuous deployment across distributed infrastructure while monitoring model performance at scale. Edge implementations need robust update mechanisms and health monitoring across potentially thousands of devices. Mobile systems require seamless app updates and performance monitoring without disrupting user experience. Tiny ML faces unique challenges in deploying updates to embedded devices while ensuring continuous operation. Across all scales, the fundamental problems of deployment, monitoring, and maintenance remain consistent, even as solutions vary.

Trustworthy AI considerations ensure ML systems operate reliably, securely, and with appropriate privacy protections. Cloud implementations must secure massive amounts of data while ensuring model predictions remain reliable at scale. Edge systems need to protect local data processing while maintaining model accuracy in diverse environments. Mobile ML must preserve user privacy while delivering consistent performance. Tiny ML systems, despite their size, must still ensure secure operation and reliable inference. These trust-

[23] GPU Clusters: Groups of GPUs networked together to provide increased processing power for tasks like model training.

worthiness considerations cut across all implementations, reflecting the critical importance of building ML systems that users can depend on.

The progression through these layers, from diverse implementations through core principles to shared considerations, reveals why ML systems can be studied as a unified field despite their apparent differences. While specific solutions may vary dramatically based on scale and context, the fundamental challenges remain remarkably consistent. This understanding becomes particularly valuable as we move toward increasingly sophisticated hybrid systems that combine multiple implementation approaches.

The convergence of fundamental principles across ML implementations helps explain why hybrid approaches work so effectively in practice. As we saw in our discussion of hybrid ML, different implementations naturally complement each other precisely because they share these core foundations. Whether we're looking at train-serve splits that leverage cloud resources for training and edge devices for inference, or hierarchical processing that combines Tiny ML sensors with edge aggregation and cloud analytics, the shared principles enable seamless integration across scales.

2.7.4 Principles to Practice

This convergence also suggests why techniques and insights often transfer well between different scales of ML systems. A deep understanding of data pipelines in cloud environments can inform how we structure data flow in embedded systems. Resource management strategies developed for mobile devices might inspire new approaches to cloud optimization. System architecture patterns that prove effective at one scale often adapt surprisingly well to others.

Understanding these fundamental principles and shared considerations provides a foundation for comparing different ML implementations more effectively. While each approach has its distinct characteristics and optimal use cases, they all build upon the same core elements. As we move into our detailed comparison in the next section, keeping these shared foundations in mind will help us better appreciate both the differences and similarities between various ML system implementations.

2.8 System Comparison

Building on the shared principles explored earlier, we can synthesize our understanding by examining how the various ML system approaches compare across different dimensions. This synthesis highlights the trade-offs system designers often face when choosing deployment options and how these decisions align with core principles like resource management, data pipelines, and system architecture.

The relationship between computational resources and deployment location forms one of the most fundamental comparisons across ML systems. As we move from cloud deployments to tiny devices, we observe a dramatic reduction in available computing power, storage, and energy consumption. Cloud ML systems, with their data center infrastructure, can leverage virtually unlimited resources, processing data at the scale of petabytes and training models with billions of parameters. Edge ML systems, while more constrained, still offer significant computational capability through specialized hardware like edge GPUs and neural processing units. Mobile ML represents a middle ground, balancing computational power with energy efficiency on devices like smartphones and tablets. At the far end of the spectrum, TinyML operates under severe resource constraints, often limited to kilobytes of memory and milliwatts of power consumption.

Table 2.2: Comparison of feature aspects across Cloud ML, Edge ML, and Tiny ML.

Aspect	Cloud ML	Edge ML	Mobile ML	Tiny ML
Performance				
Processing Location	Centralized cloud servers (Data Centers)	Local edge devices (gateways, servers)	Smartphones and tablets	Ultra-low-power microcontrollers and embedded systems
Latency	High (100 ms-1000 ms+)	Moderate (10-100 ms)	Low-Moderate (5-50 ms)	Very Low (1-10 ms)
Compute Power	Very High (Multiple GPUs/TPUs)	High (Edge GPUs)	Moderate (Mobile NPUs/GPUs)	Very Low (MCU/tiny processors)
Storage Capacity	Unlimited (petabytes+)	Large (terabytes)	Moderate (gigabytes)	Very Limited (kilobytes-megabytes)
Energy Consumption	Very High (kW-MW range)	High (100 s W)	Moderate (1-10 W)	Very Low (mW range)
Scalability	Excellent (virtually unlimited)	Good (limited by edge hardware)	Moderate (per-device scaling)	Limited (fixed hardware)
Operational				
Data Privacy	Basic-Moderate (Data leaves device)	High (Data stays in local network)	High (Data stays on phone)	Very High (Data never leaves sensor)
Connectivity Required	Constant high-bandwidth	Intermittent	Optional	None
Offline Capability	None	Good	Excellent	Complete
Real-time Processing	Dependent on network	Good	Very Good	Excellent
Deployment				
Cost	High ($1000s+/month)	Moderate ($100s-1000s)	Low ($0-10s)	Very Low ($1-10s)
Hardware Requirements	Cloud infrastructure	Edge servers/gateways	Modern smartphones	MCUs/embedded systems
Development Complexity	High (cloud expertise needed)	Moderate-High (edge+networking)	Moderate (mobile SDKs)	High (embedded expertise)
Deployment Speed	Fast	Moderate	Fast	Slow

The operational characteristics of these systems reveal another important dimension of comparison. Table 2.2 organizes these characteristics into logical groupings, highlighting performance, operational considerations, costs, and development aspects. For instance, latency shows a clear gradient: cloud systems typically incur delays of 100-1000 ms due to network communication, while edge systems reduce this to 10-100 ms by processing data locally. Mobile

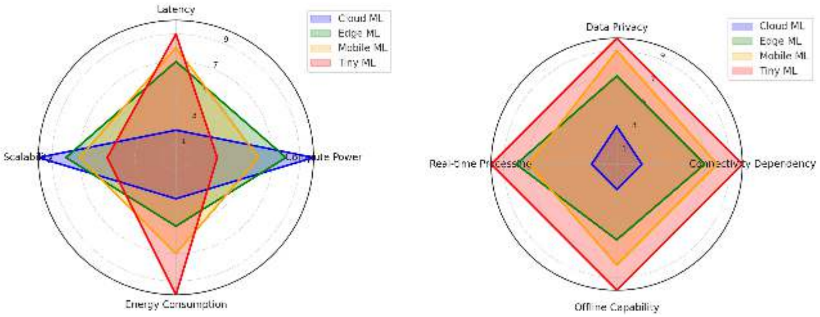
ML achieves even lower latencies of 5-50 ms for many tasks, and TinyML systems can respond in 1-10 ms for simple inferences. Similarly, privacy and data handling improve progressively as computation shifts closer to the data source, with TinyML offering the strongest guarantees by keeping data entirely local to the device.

The table is designed to provide a high-level view of how these paradigms differ across key dimensions, making it easier to understand the trade-offs and select the most appropriate approach for specific deployment needs.

To complement the details presented in Table 2.2, radar plots are presented below. These visualizations highlight two critical dimensions: performance characteristics and operational characteristics. The performance characteristics plot in Figure 2.12 focuses on latency, compute power, energy consumption, and scalability. As discussed earlier, Cloud ML demands exceptional compute power and demonstrates good scalability, making it ideal for large-scale tasks requiring extensive resources. Tiny ML, in contrast, excels in latency and energy efficiency due to its lightweight and localized processing, suitable for low-power, real-time scenarios. Edge ML and Mobile ML strike a balance, offering moderate scalability and efficiency for a variety of applications.

Figure 2.12: a) Performance characteristics. b) Operational characteristics.

Figure 2.13:



The operational characteristics plot in Figure 2.13 emphasizes data privacy, connectivity independence, offline capability, and real-time processing. Tiny ML emerges as a highly independent and private paradigm, excelling in offline functionality and real-time responsiveness. In contrast, Cloud ML relies on centralized infrastructure and constant connectivity, which can be a limitation in scenarios demanding autonomy or low-latency decision-making.

Development complexity and deployment considerations also vary significantly across these paradigms. Cloud ML benefits from mature development tools and frameworks but requires expertise in cloud infrastructure. Edge ML demands knowledge of both ML and networking protocols, while Mobile ML developers must understand mobile-specific optimizations and platform constraints. TinyML development, though targeting simpler devices, often requires specialized knowledge of embedded systems and careful optimization to work within severe resource constraints.

Cost structures differ markedly as well. Cloud ML typically involves ongoing operational costs for computation and storage, often running into thousands of dollars monthly for large-scale deployments. Edge ML requires significant upfront investment in edge devices but may reduce ongoing costs. Mobile ML leverages existing consumer devices, minimizing additional hardware costs, while TinyML solutions can be deployed for just a few dollars per device, though development costs may be higher.

These comparisons reveal that each paradigm has distinct advantages and limitations. Cloud ML excels at complex, data-intensive tasks but requires constant connectivity. Edge ML offers a balance of computational power and local processing. Mobile ML provides personalized intelligence on ubiquitous devices. TinyML enables ML in previously inaccessible contexts but requires careful optimization. Understanding these trade-offs is crucial for selecting the appropriate deployment strategy for specific applications and constraints.

2.9 Deployment Decision Framework

We have examined the diverse paradigms of machine learning systems, including Cloud ML, Edge ML, Mobile ML, and Tiny ML, each with its own characteristics, trade-offs, and use cases. Selecting an optimal deployment strategy requires careful consideration of multiple factors.

To facilitate this decision-making process, we present a structured framework in Figure 2.14. This framework distills the chapter's key insights into a systematic approach for determining the most suitable deployment paradigm based on specific requirements and constraints.

The framework is organized into five fundamental layers of consideration:

- **Privacy**: Determines whether processing can occur in the cloud or must remain local to safeguard sensitive data.
- **Latency**: Evaluates the required decision-making speed, particularly for real-time or near-real-time processing needs.
- **Reliability**: Assesses network stability and its impact on deployment feasibility.
- **Compute Needs**: Identifies whether high-performance infrastructure is required or if lightweight processing suffices.
- **Cost and Energy Efficiency**: Balances resource availability with financial and energy constraints, particularly crucial for low-power or budget-sensitive applications.

As designers progress through these layers, each decision point narrows the viable options, ultimately guiding them toward one of the four deployment paradigms. This systematic approach proves valuable across various scenarios. For instance, privacy-sensitive healthcare applications might prioritize local processing over cloud solutions, while high-performance recommendation engines typically favor cloud infrastructure. Similarly, applications requiring real-time responses often gravitate toward edge or mobile-based deployment.

While not exhaustive, this framework provides a practical roadmap for navigating deployment decisions. By following this structured approach, system designers can evaluate trade-offs and align their deployment choices with technical, financial, and operational priorities, even as they address the unique challenges of each application.

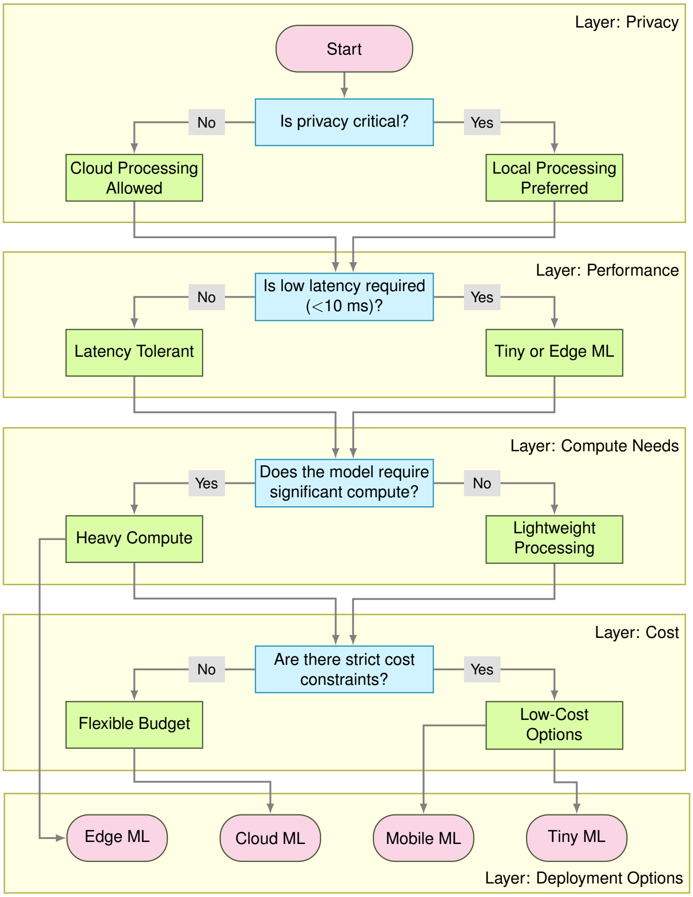


Figure 2.14: A decision flowchart for selecting the most suitable ML deployment paradigm.

2.10 Conclusion

This chapter has explored the diverse landscape of machine learning systems, highlighting their unique characteristics, benefits, challenges, and applications. Cloud ML leverages immense computational resources, excelling in large-scale data processing and model training but facing limitations such as latency and privacy concerns. Edge ML bridges this gap by enabling localized processing, reducing latency, and enhancing privacy. Mobile ML builds on these strengths, harnessing the ubiquity of smartphones to provide responsive, user-centric applications. At the smallest scale, Tiny ML extends the reach of machine learning to resource-constrained devices, opening new domains of application.

Together, these paradigms reflect an ongoing progression in machine learning, moving from centralized systems in the cloud to increasingly distributed and specialized deployments across edge, mobile, and tiny devices. This evolution marks a shift toward systems that are finely tuned to specific deployment contexts, balancing computational power, energy efficiency, and real-time responsiveness. As these paradigms mature, hybrid approaches are emerging, blending their strengths to unlock new possibilities—from cloud-based training paired with edge inference to federated learning[24] and hierarchical processing[25].

Despite their variety, ML systems can be distilled into a core set of unifying principles that span resource management, data pipelines, and system architecture. These principles provide a structured framework for understanding and designing ML systems at any scale. By focusing on these shared fundamentals and mastering their design and optimization, we can navigate the complexity of the ML landscape with clarity and confidence. As we continue to advance, these principles will act as a compass, guiding our exploration and innovation within the ever-evolving field of machine learning systems. Regardless of how diverse or complex these systems become, a strong grasp of these foundational concepts will remain essential to unlocking their full potential.

2.11 Resources

i Slides

- Embedded Systems Overview.
- Embedded Computer Hardware.
- Embedded I/O.
- Embedded systems software.
- Embedded ML software.
- Embedded Inference.
- Tiny ML on Microcontrollers.
- Tiny ML as a Service (Tiny MLaaS):

—Tiny MLaaS: Introduction.
—Tiny MLaaS: Design Overview.

! Videos

- *Coming soon.*

Exercises

To reinforce the concepts covered in this chapter, we have curated a set of exercises that challenge students to apply their knowledge and deepen their understanding.

- *Coming soon.*

[24] Federated learning is an approach where global models are trained locally on devices and then aggregated back on a server, maintaining user privacy.

[25] Hierarchical processing refers to analyzing and processing data in a hierarchical manner, often to manage computational complexity.

Chapter 3

DL Primer

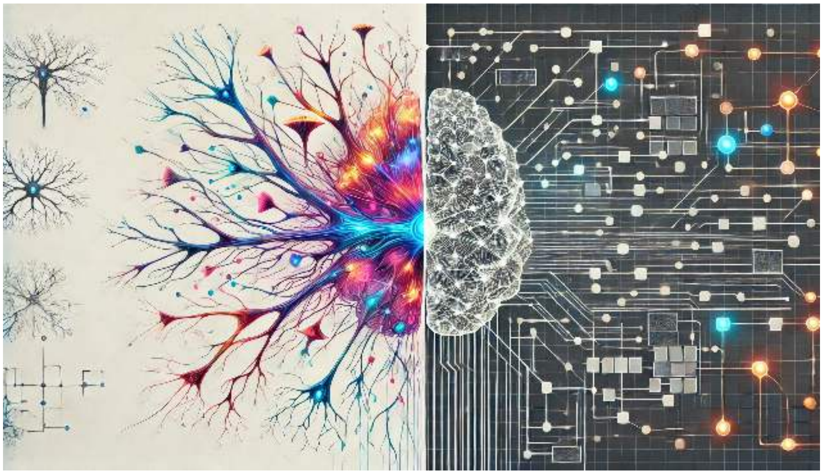


Figure 3.1: *DALL·E 3 Prompt: A rectangular illustration divided into two halves on a clean white background. The left side features a detailed and colorful depiction of a biological neural network, showing interconnected neurons with glowing synapses and dendrites. The right side displays a sleek and modern artificial neural network, represented by a grid of interconnected nodes and edges resembling a digital circuit. The transition between the two sides is distinct but harmonious, with each half clearly illustrating its respective theme: biological on the left and artificial on the right.*

Purpose

What inspiration from nature drives the development of machine learning systems, and how do biological neural processes inform their fundamental design?

The neural systems of nature offer profound insights into information processing and adaptation, inspiring the core principles of modern machine learning. Translating biological mechanisms into computational frameworks illuminates fundamental patterns that shape artificial neural networks. These patterns reveal essential relationships between biological principles and their digital counterparts, establishing building blocks for understanding more complex architectures. Analyzing these mappings from natural to artificial provides critical insights into system design, laying the foundation for exploring advanced neural architectures and their practical implementations.

💡 Learning Objectives

- Understand the biological inspiration for artificial neural networks and how this foundation informs their design and function.
- Explore the fundamental structure of neural networks, including neurons, layers, and connections.
- Examine the processes of forward propagation, backward propagation, and optimization as the core mechanisms of learning.
- Understand the complete machine learning pipeline, from preprocessing through neural computation to post-processing.
- Compare and contrast training and inference phases, understanding their distinct computational requirements and optimizations.
- Learn how neural networks process data to extract patterns and make predictions, bridging theoretical concepts with computational implementations.

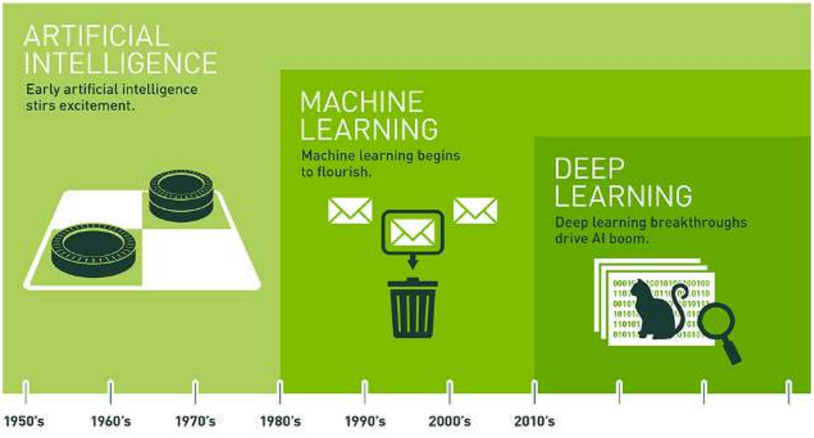
3.1 Overview

Neural networks, a foundational concept within machine learning and artificial intelligence, are computational models inspired by the structure and function of biological neural systems. These networks represent a critical intersection of algorithms, mathematical frameworks, and computing infrastructure, making them integral to solving complex problems in AI.

When studying neural networks, it is helpful to place them within the broader hierarchy of AI and machine learning. Figure 3.2 provides a visual representation of this context. AI, as the overarching field, encompasses all computational methods that aim to mimic human cognitive functions. Within AI, machine learning includes techniques that enable systems to learn patterns from data. Neural networks, a key subset of ML, form the backbone of more advanced learning systems, including deep learning, by modeling complex relationships in data through interconnected computational units.

The emergence of neural networks reflects key shifts in how AI systems process information across three fundamental dimensions:

- **Data**: From manually structured and rule-based datasets to raw, high-dimensional data. Neural networks are particularly adept at learning from complex and unstructured data, making them essential for tasks involving images, speech, and text.
- **Algorithms**: From explicitly programmed rules to adaptive systems capable of learning patterns directly from data. Neural networks eliminate the need for manual feature engineering by discovering representations automatically through layers of interconnected units.
- **Computation**: From simple, sequential operations to massively parallel computations. The scalability of neural networks has driven demand for advanced hardware, such as GPUs, that can efficiently process large models and datasets.


ARTIFICIAL INTELLIGENCE
Early artificial intelligence stirs excitement.
MACHINE LEARNING
Machine learning begins to flourish.
DEEP LEARNING
Deep learning breakthroughs drive AI boom.
1950's
1960's
1970's
1980's
1990's
2000's
2010's

Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

Figure 3.2: The diagram illustrates artificial intelligence as the overarching field encompassing all computational methods that mimic human cognitive functions. Machine learning is a subset of AI that includes algorithms capable of learning from data. Deep learning, a further subset of ML, specifically involves neural networks that are able to learn more complex patterns in large volumes of data. Source: NVIDIA.

These shifts emphasize the importance of understanding neural networks, not only as mathematical constructs but also as practical components of real-world AI systems. The development and deployment of neural networks require careful consideration of computational efficiency, data processing workflows, and hardware optimization. To build a strong foundation, this chapter focuses on the core principles of neural networks, exploring their structure, functionality, and learning mechanisms. By understanding these basics, readers will be well-prepared to delve into more advanced architectures and their systems-level implications in later chapters.

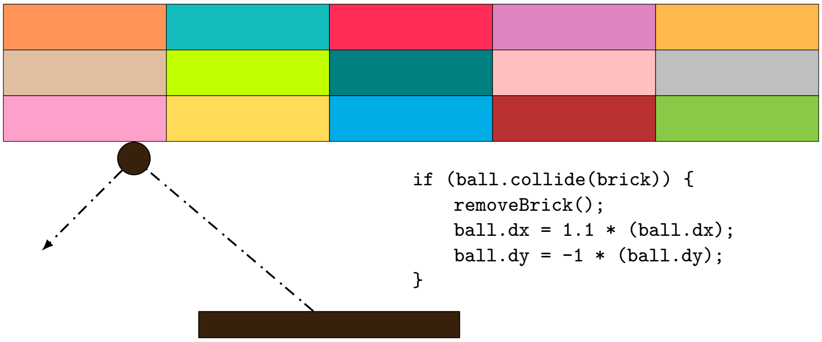
3.2 The Evolution to Deep Learning

The current era of AI represents a transformative advance in computational problem-solving, marking the latest stage in an evolution from rule-based programming through classical machine learning to modern neural networks. To understand its significance, we must trace this progression and examine how each approach builds upon and addresses the limitations of its predecessors.

3.2.1 Rule-Based Programming

Traditional programming requires developers to explicitly define rules that tell computers how to process inputs and produce outputs. Consider a simple game like Breakout, shown in Figure 3.3. The program needs explicit rules for every interaction: when the ball hits a brick, the code must specify that the brick should be removed and the ball's direction should be reversed. While this approach works well for games with clear physics and limited states, it demonstrates an inherent limitation of rule-based systems.

This rules-based paradigm extends to all traditional programming, as illustrated in Figure 3.4. The program takes both rules for processing and input data to produce outputs. Early artificial intelligence research explored whether



```
if (ball.collide(brick)) {
    removeBrick();
    ball.dx = 1.1 * (ball.dx);
    ball.dy = -1 * (ball.dy);
}
```

Figure 3.3: Rule-based programming.

this approach could scale to solve complex problems by encoding sufficient rules to capture intelligent behavior.

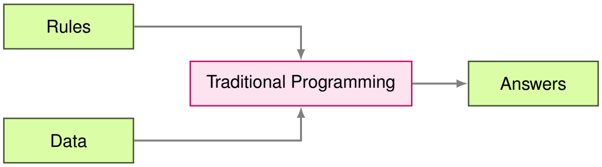
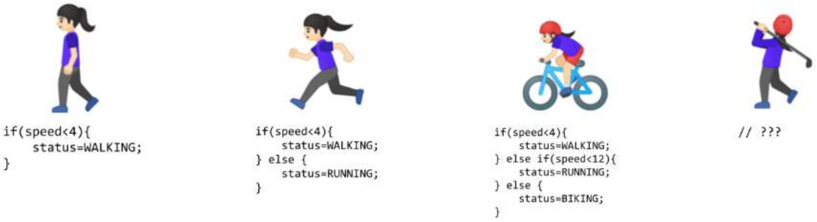


Figure 3.4: Traditional programming.

However, the limitations of rule-based approaches become evident when addressing complex real-world tasks. Consider the problem of recognizing human activities, shown in Figure 3.5. Initial rules might appear straightforward: classify movement below 4 mph as walking and faster movement as running. Yet real-world complexity quickly emerges. The classification must account for variations in speed, transitions between activities, and numerous edge cases. Each new consideration requires additional rules, leading to increasingly complex decision trees.



```
if(speed<4){
    status=WALKING;
}
```

```
if(speed<4){
    status=WALKING;
} else {
    status=RUNNING;
}
```

```
if(speed<4){
    status=WALKING;
} else if(speed<12){
    status=RUNNING;
} else {
    status=BIKING;
}
```

```
// ???
```

Figure 3.5: Activity rules.

This challenge extends to computer vision tasks. Detecting objects like cats in images would require rules about System Implications: pointed ears, whiskers, typical body shapes. Such rules would need to account for variations in viewing angle, lighting conditions, partial occlusions, and natural variations among instances. Early computer vision systems attempted this approach through geometric rules but achieved success only in controlled environments with well-defined objects.

This knowledge engineering approach[0] characterized artificial intelligence

[0] Knowledge Engineering: The process of creating rules and heuristics for problem-solving and decision-making within artificial intelligence systems.

research in the 1970s and 1980s. Expert systems[1] encoded domain knowledge as explicit rules, showing promise in specific domains with well-defined parameters but struggling with tasks humans perform naturally, such as object recognition, speech understanding, or natural language interpretation. These limitations highlighted a fundamental challenge: many aspects of intelligent behavior rely on implicit knowledge that resists explicit rule-based representation.

3.2.2 Classical Machine Learning

The limitations of pure rule-based systems led researchers to explore approaches that could learn from data. Machine learning offered a promising direction: instead of writing rules for every situation, we could write programs that found patterns in examples. However, the success of these methods still depended heavily on human insight to define what patterns might be important, a process known as feature engineering[2].

Feature engineering involves transforming raw data into representations that make patterns more apparent to learning algorithms. In computer vision, researchers developed sophisticated methods to extract meaningful patterns from images. The Histogram of Oriented Gradients (HOG) method, shown in Figure 3.6, exemplifies this approach. HOG works by first identifying edges in an image, which are places where brightness changes sharply and often indicate object boundaries. It then divides the image into small cells and measures how edges are oriented within each cell, summarizing these orientations in a histogram. This transformation converts raw pixel values into a representation that captures important shape information while being robust to variations in lighting and small changes in position.

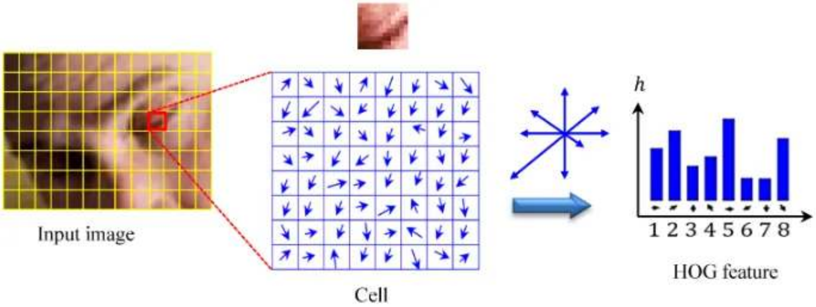


Figure 3.6: Histogram of Oriented Gradients (HOG) requires explicit feature engineering.

Other feature extraction methods like SIFT (Scale-Invariant Feature Transform) and Gabor filters provided different ways to capture patterns in images. SIFT found distinctive points that could be recognized even when an object's size or orientation changed. Gabor filters helped identify textures and repeated patterns. Each method encoded different types of human insight about what makes visual patterns recognizable.

These engineered features enabled significant advances in computer vision during the 2000s. Systems could now recognize objects with some robustness to real-world variations, leading to applications in face detection, pedestrian

[1] Expert systems: An AI program that leverages expert knowledge in a particular field to answer questions or solve problems.

[2] Feature engineering: The process of using domain knowledge to create features that make machine learning algorithms work.

detection, and object recognition. However, the approach had fundamental limitations. Experts needed to carefully design feature extractors for each new problem, and the resulting features might miss important patterns that weren't anticipated in their design.

3.2.3 Neural Networks and Representation Learning

Neural networks represent a fundamental shift in how we approach problem solving with computers, establishing a new programming paradigm that learns from data rather than following explicit rules. This shift becomes particularly evident when considering tasks like computer vision—specifically, identifying objects in images.

> ℹ Definition of Deep Learning
>
> **Deep Learning** is a *subfield* of machine learning that utilizes *artificial neural networks with multiple layers* to *automatically learn hierarchical representations* from data. This approach enables the extraction of *complex patterns* from large datasets, facilitating tasks like *image recognition, natural language processing, and speech recognition* without explicit feature engineering. Deep learning's effectiveness arises from its ability to *learn features directly* from raw data, *adapt to diverse data structures*, and *scale with increasing data volume*.

Unlike traditional programming approaches that require manual rule specification, deep learning utilizes artificial neural networks with multiple layers to automatically learn hierarchical representations from data. This enables systems to extract complex patterns from large datasets, facilitating tasks like image recognition, natural language processing, and speech recognition without explicit feature engineering. The effectiveness of deep learning comes from its ability to learn features directly from raw data, adapt to diverse data structures, and scale with increasing data volume.

Deep learning fundamentally differs by learning directly from raw data. Traditional programming, as we saw earlier in Figure 3.4, required both rules and data as inputs to produce answers. Machine learning inverts this relationship, as shown in Figure 3.7. Instead of writing rules, we provide examples (data) and their correct answers to discover the underlying rules automatically. This shift eliminates the need for humans to specify what patterns are important.

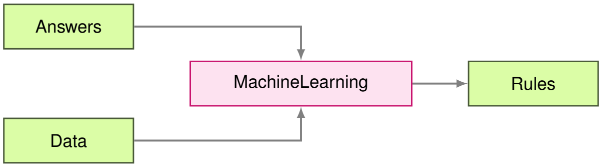


Figure 3.7: Deep learning.

The system discovers these patterns automatically from examples. When shown millions of images of cats, the system learns to identify increasingly complex visual patterns, from simple edges to more sophisticated combinations that make up cat-like features. This mirrors how our own visual system works, building up understanding from basic visual elements to complex objects.

Unlike traditional approaches where performance often plateaus with more data and computation, deep learning systems continue to improve as we provide more resources. More training examples help the system recognize more variations and nuances. More computational power enables the system to discover more subtle patterns. This scalability has led to dramatic improvements in performance; for example, the accuracy of image recognition systems has improved from 74% in 2012 to over 95% today.

This different approach has profound implications for how we build AI systems. Deep learning's ability to learn directly from raw data eliminates the need for manual feature engineering, but it comes with new demands. We need sophisticated infrastructure to handle massive datasets, powerful computers to process this data, and specialized hardware to perform the complex mathematical calculations efficiently. The computational requirements of deep learning have even driven the development of new types of computer chips optimized for these calculations.

The success of deep learning in computer vision exemplifies how this approach, when given sufficient data and computation, can surpass traditional methods. This pattern has repeated across many domains, from speech recognition to game playing, establishing deep learning as a transformative approach to artificial intelligence.

3.2.4 Neural System Implications

The progression from traditional programming to deep learning represents not just a shift in how we solve problems, but a fundamental transformation in computing system requirements. This transformation becomes particularly critical when we consider the full spectrum of ML systems, from massive cloud deployments to resource-constrained tiny ML devices.

Traditional programs follow predictable patterns. They execute sequential instructions, access memory in regular patterns, and utilize computing resources in well-understood ways. A typical rule-based image processing system might scan through pixels methodically, applying fixed operations with modest and predictable computational and memory requirements. These characteristics made traditional programs relatively straightforward to deploy across different computing platforms.

Machine learning with engineered features introduced new complexities. Feature extraction algorithms required more intensive computation and structured data movement. The HOG feature extractor discussed earlier, for instance, requires multiple passes over image data, computing gradients and constructing histograms. While this increased both computational demands and memory complexity, the resource requirements remained relatively predictable and scalable across platforms.

Deep learning, however, fundamentally reshapes system requirements across multiple dimensions. Table 3.1 shows the evolution of system requirements across programming paradigms:

Table 3.1: Evolution of system requirements across programming paradigms.

System Aspect	Traditional Programming	ML with Features	Deep Learning
Computation	Sequential, predictable paths	Structured parallel operations	Massive matrix parallelism
Memory Access	Small, predictable patterns	Medium, batch-oriented	Large, complex hierarchical patterns
Data Movement	Simple input/output flows	Structured batch processing	Intensive cross-system movement
Hardware Needs	CPU-centric	CPU with vector units	Specialized accelerators
Resource Scaling	Fixed requirements	Linear with data size	Exponential with complexity

These differences manifest in several critical ways, with implications across the entire ML systems spectrum.

3.2.4.1 Computation Patterns

While traditional programs follow sequential logic flows, deep learning requires massive parallel operations on matrices. This shift explains why conventional CPUs, designed for sequential processing, prove inefficient for neural network computations. The need for parallel processing has driven the adoption of specialized hardware architectures, ranging from powerful cloud GPUs to specialized mobile processors to tiny ML accelerators.

3.2.4.2 Memory Systems

Traditional programs typically maintain small, fixed memory footprints. Deep learning models, however, must manage parameters across complex memory hierarchies. Memory bandwidth often becomes the primary performance bottleneck, creating particular challenges for resource-constrained systems. This drives different optimization strategies across the ML systems spectrum, ranging from memory-rich cloud deployments to heavily optimized tiny ML implementations.

3.2.4.3 System Scaling

Perhaps most importantly, deep learning fundamentally changes how systems scale and the critical importance of efficiency. Traditional programs have relatively fixed resource requirements with predictable performance characteristics. Deep learning systems, however, can consume exponentially more resources as models grow in complexity. This relationship between model capability and resource consumption makes system efficiency a central concern.

The need to bridge algorithmic concepts with hardware realities becomes crucial. While traditional programs map relatively straightforwardly to standard computer architectures, deep learning requires us to think carefully about:

- How to efficiently map matrix operations to physical hardware
- Ways to minimize data movement across memory hierarchies
- Methods to balance computational capability with resource constraints
- Techniques to optimize both algorithm and system-level efficiency

[3] Hardware accelerators: Specialized hardware designed to perform certain types of operations more efficiently than general-purpose computing units.

These fundamental shifts explain why deep learning has spurred innovations across the entire computing stack. From specialized hardware accelerators[3] to new memory architectures[4] to sophisticated software frameworks, the demands of deep learning continue to reshape computer system design. Interestingly, many of these challenges, efficiency, scaling, and adaptability, are ones that biological systems have already solved. This brings us to a critical question: what can we learn from nature's own information processing system and strive to mimic them as artificially intelligent systems.

3.3 Biological to Artificial Neurons

The quest to create artificial intelligence has been profoundly influenced by our understanding of biological intelligence, particularly the human brain. This isn't surprising; the brain represents the most sophisticated information processing system we know of. It is capable of learning, adapting, and solving complex problems while maintaining remarkable energy efficiency. The way our brains function has provided fundamental insights that continue to shape how we approach artificial intelligence.

3.3.1 Biological Intelligence

When we observe biological intelligence, several key principles emerge. The brain demonstrates an extraordinary ability to learn from experience, constantly modifying its neural connections based on new information and interactions with the environment. This adaptability is fundamental; every experience potentially alters the brain's structure and refines its responses for future situations. This biological capability directly inspired one of the core principles of machine learning: the ability to learn and improve from data rather than following fixed, pre-programmed rules.

Another striking feature of biological intelligence is its parallel processing capability. The brain processes vast amounts of information simultaneously, with different regions specializing in specific functions while working in concert. This distributed, parallel architecture stands in stark contrast to traditional sequential computing and has significantly influenced modern AI system design. The brain's ability to efficiently coordinate these parallel processes while maintaining coherent function represents a level of sophistication we're still working to fully understand and replicate.

The brain's pattern recognition capabilities are particularly noteworthy. Biological systems excel at identifying patterns in complex, noisy data, whether it is recognizing faces in a crowd, understanding speech in a noisy environment, or identifying objects from partial information. This remarkable ability has inspired numerous AI applications, particularly in computer vision and speech recognition systems. The brain accomplishes these tasks with an efficiency that artificial systems are still striving to match.

Perhaps most remarkably, biological systems achieve all this with incredible energy efficiency. The human brain operates on approximately 20 watts of power, about the same as a low-power light bulb, while performing complex

[4] Memory architecture: The design of a computer's memory system, including the physical structure and components, data organization and access, and pathways between memory and computing units.

cognitive tasks that would require orders of magnitude more power in current artificial systems. This efficiency hasn't just impressed researchers; it has become a crucial goal in the development of AI hardware and algorithms.

These biological principles have led to two distinct but complementary approaches in artificial intelligence. The first attempts to directly mimic neural structure and function, leading to artificial neural networks and deep learning architectures that structurally resemble biological neural networks. The second takes a more abstract approach, adapting biological principles to work efficiently within the constraints of computer hardware without necessarily copying biological structures exactly. In the following sections, we will explore how these approaches manifest in practice, beginning with the fundamental building block of neural networks: the neuron itself.

3.3.2 Transition to Artificial Neurons

To understand how biological principles translate into artificial systems, we must first examine the basic unit of biological information processing: the neuron. This cellular building block provides the blueprint for its artificial counterpart and helps us understand how complex neural networks emerge from simple components working in concert.

In biological systems, the neuron (or cell) is the basic functional unit of the nervous system. Understanding its structure is crucial before we draw parallels to artificial systems. Figure 3.8 illustrates the structure of a biological neuron.

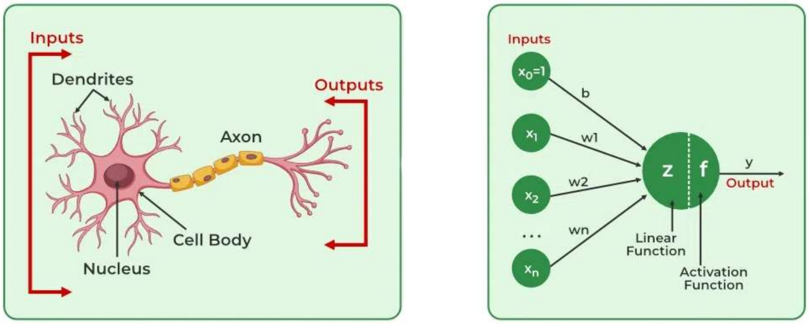


Figure 3.8: Bilogical structure of a neuron and its mapping to an artificial neuron. Source: Geeksforgeeks

A biological neuron consists of several key components. The central part is the cell body, or soma, which contains the nucleus and performs the cell's basic life processes. Extending from the soma are branch-like structures called dendrites, which receive signals from other neurons. At the junctions where signals are passed between neurons are synapses. Finally, a long, slender projection called the axon conducts electrical impulses away from the cell body to other neurons.

The neuron functions as follows: Dendrites receive inputs from other neurons, with synapses determining the strength of the connections. The soma integrates these signals and decides whether to trigger an output signal. If triggered, the axon transmits this signal to other neurons.

Each element of a biological neuron has a computational analog in artificial systems, reflecting the principles of learning, adaptability, and efficiency found in nature. To better understand how biological intelligence informs artificial systems, Table 3.2 captures the mapping between the components of biological and artificial neurons. This should be viewed alongside Figure 3.8 for a complete picture. Together, they paint a picture of the biological-to-artificial neuron mapping.

Table 3.2: Mapping the biological neuron structure to an artificial neuron.

Biological Neuron	Artificial Neuron
Cell	Neuron / Node
Dendrites / Synapse	Weights
Soma	Net Input
Axon	Output

Each component serves a similar function, albeit through vastly different mechanisms. Here, we explain these mappings and their implications for artificial neural networks.

1. **Cell** ⟷ **Neuron/Node**: The artificial neuron or node serves as the fundamental computational unit, mirroring the cell's role in biological systems.
2. **Dendrites/Synapse** ⟷ **Weights**: Weights in artificial neurons represent connection strengths, analogous to synapses in biological neurons. These weights are adjustable, enabling learning and optimization over time.
3. **Soma** ⟷ **Net Input**: The net input in artificial neurons sums weighted inputs to determine activation, similar to how the soma integrates signals in biological neurons.
4. **Axon** ⟷ **Output**: The output of an artificial neuron passes processed information to subsequent network layers, much like an axon transmits signals to other neurons.

This mapping illustrates how artificial neural networks simplify and abstract biological processes while preserving their essential computational principles. However, understanding individual neurons is just the beginning—the true power of neural networks emerges from how these basic units work together in larger systems.

3.3.3 Artificial Intelligence

The translation from biological principles to artificial computation requires a deep appreciation of what makes biological neural networks so effective at both the cellular and network levels. The brain processes information through distributed computation across billions of neurons, each operating relatively slowly compared to silicon transistors. A biological neuron fires at approximately 200Hz, while modern processors operate at gigahertz frequencies. Despite this speed limitation, the brain's parallel architecture enables sophisticated real-time processing of complex sensory input, decision making, and control of behavior.

This computational efficiency emerges from the brain's basic organizational principles. Each neuron acts as a simple processing unit, integrating inputs from thousands of other neurons and producing a binary output signal based on whether this integrated input exceeds a threshold. The connection strengths between neurons, mediated by synapses, are continuously modified through experience. This synaptic plasticity[5] forms the basis for learning and adaptation in biological neural networks. These biological principles suggest key computational elements needed in artificial neural systems:

- Simple processing units that integrate multiple inputs
- Adjustable connection strengths between units
- Nonlinear activation based on input thresholds
- Parallel processing architecture
- Learning through modification of connection strengths

3.3.4 Computational Translation

We face the challenge of capturing the essence of neural computation within the rigid framework of digital systems. The implementation of biological principles in artificial neural systems represents a nuanced balance between biological fidelity and computational efficiency. At its core, an artificial neuron captures the essential computational properties of its biological counterpart through mathematical operations that can be efficiently executed on digital hardware.

Table 3.3 provides a systematic view of how key biological features map to their computational counterparts. Each biological feature has an analog in computational systems, revealing both the possibilities and limitations of digital neural implementation, which we will learn more about later.

Table 3.3: Translating biological features to the computing domain.

Biological Feature	Computational Translation
Neuron firing	Activation function
Synaptic strength	Weighted connections
Signal integration	Summation operation
Distributed memory	Weight matrices
Parallel processing	Concurrent computation

The basic computational unit in artificial neural networks, the artificial neuron, simplifies the complex electrochemical processes of biological neurons into three fundamental operations. First, input signals are weighted, mimicking how biological synapses modulate incoming signals with different strengths. Second, these weighted inputs are summed together, analogous to how a biological neuron integrates incoming signals in its cell body. Finally, the summed input passes through an activation function that determines the neuron's output, similar to how a biological neuron fires based on whether its membrane potential exceeds a threshold.

This mathematical abstraction preserves key computational principles while enabling efficient digital implementation. The weighting of inputs allows the network to learn which connections are important, just as biological neural networks strengthen or weaken synaptic connections through experience. The summation operation captures how biological neurons integrate multiple inputs

[5] Synaptic Plasticity: The ability of connections between neurons to change in strength in response to changes in synaptic activity.

into a single decision. The activation function introduces nonlinearity essential for learning complex patterns, much like the threshold-based firing of biological neurons.

Memory in artificial neural networks takes a markedly different form from biological systems. While biological memories are distributed across synaptic connections and neural patterns, artificial networks store information in discrete weights and parameters. This architectural difference reflects the constraints of current computing hardware, where memory and processing are physically separated rather than integrated as in biological systems. Despite these implementation differences, artificial neural networks achieve similar functional capabilities in pattern recognition and learning.

The brain's massive parallelism represents a fundamental challenge in artificial implementation. While biological neural networks process information through billions of neurons operating simultaneously, artificial systems approximate this parallelism through specialized hardware like GPUs and tensor processing units. These devices efficiently compute the matrix operations that form the mathematical foundation of artificial neural networks, achieving parallel processing at a different scale and granularity than biological systems.

3.3.5 System Requirements

The computational translation of neural principles creates specific demands on the underlying computing infrastructure. These requirements emerge from the fundamental differences between biological and artificial implementations of neural processing, shaping how we design and build systems capable of supporting artificial neural networks.

Table 3.4 shows how each computational element drives particular system requirements. From this mapping, we can see how the choices made in computational translation directly influence the hardware and system architecture needed for implementation.

Table 3.4: From computation to system requirements.

Computational Element	System Requirements
Activation functions	Fast nonlinear operation units
Weight operations	High-bandwidth memory access
Parallel computation	Specialized parallel processors
Weight storage	Large-scale memory systems
Learning algorithms	Gradient computation hardware

Storage architecture represents a critical requirement, driven by the fundamental difference in how biological and artificial systems handle memory. In biological systems, memory and processing are intrinsically integrated—synapses both store connection strengths and process signals. Artificial systems, however, must maintain a clear separation between processing units and memory. This creates a need for both high-capacity storage to hold millions or billions of connection weights and high-bandwidth pathways to move this data quickly between storage and processing units. The efficiency of this data movement often becomes a critical bottleneck that biological systems do not face.

The learning process itself imposes distinct requirements on artificial systems. While biological networks modify synaptic strengths through local chemical processes, artificial networks must coordinate weight updates across the entire network. This creates substantial computational and memory demands during training—systems must not only store current weights but also maintain space for gradients and intermediate calculations. The requirement to backpropagate error signals[6], with no real biological analog, further complicates the system architecture.

Energy efficiency emerges as a final critical requirement, highlighting perhaps the starkest contrast between biological and artificial implementations. The human brain's remarkable energy efficiency, which operates on approximately 20 watts, stands in sharp contrast to the substantial power demands of artificial neural networks. Current systems often require orders of magnitude more energy to implement similar capabilities. This gap drives ongoing research in more efficient hardware architectures and has profound implications for the practical deployment of neural networks, particularly in resource-constrained environments like mobile devices or edge computing systems.

3.3.6 Evolution and Impact

We can now better appreciate how the field of deep learning evolved to meet these challenges through advances in hardware and algorithms. This journey began with early artificial neural networks in the 1950s, marked by the introduction of the Perceptron. While groundbreaking in concept, these early systems were severely limited by the computational capabilities of their era—primarily mainframe computers that lacked both the processing power and memory capacity needed for complex networks.

The development of backpropagation algorithms in the 1980s (Rumelhart, Hinton, and Williams 1986), which we will learn about later, represented a theoretical breakthrough and povided a systematic way to train multi-layer networks. However, the computational demands of this algorithm far exceeded available hardware capabilities. Training even modest networks could take weeks, making experimentation and practical applications challenging. This mismatch between algorithmic requirements and hardware capabilities contributed to a period of reduced interest in neural networks.

The term "deep learning" gained prominence in the 2010s, coinciding with significant advances in computational power and data accessibility. The field has since experienced exponential growth, as illustrated in Figure 3.9. The graph reveals two remarkable trends: computational capabilities measured in the number of Floating Point Operations per Second (FLOPS)[7] initially followed a $1.4\times$ improvement pattern from 1952 to 2010, then accelerated to a 3.4-month doubling cycle from 2012 to 2022. Perhaps more striking is the emergence of large-scale models between 2015 and 2022 (not explicitly shown or easily seen in the figure), which scaled 2 to 3 orders of magnitude faster than the general trend, following an aggressive 10-month doubling cycle.

[6] Backpropagation: A common method used to train artificial neural networks. It calculates the gradient of the loss function with respect to the weights of the network.

[7] Floating Point Operations per Second (FLOPS): A measure of computer performance, useful in fields of scientific computations that require floating-point calculations.

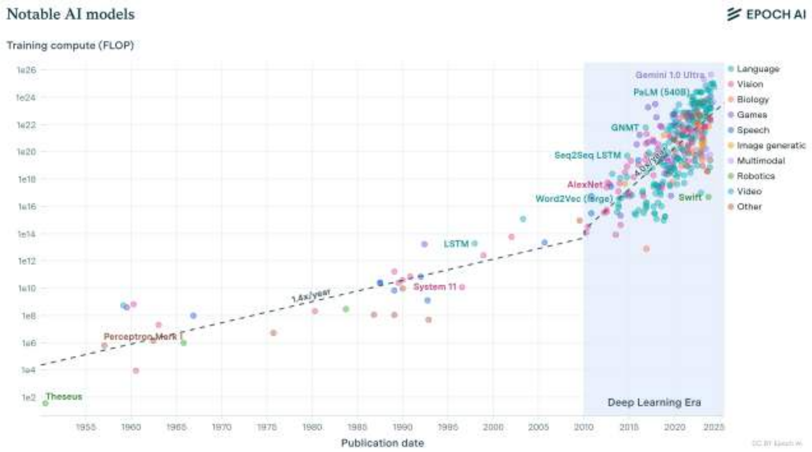


Figure 3.9: Growth of deep learning models. Source: EPOCH AI

The evolutionary trends were driven by parallel advances across three fundamental dimensions: data availability, algorithmic innovations, and computing infrastructure. These three factors, namely, data, algorithms, and infrastructure, reinforced each other in a virtuous cycle that continues to drive progress in the field today. As Figure 9.15 shows, more powerful computing infrastructure enabled processing larger datasets. Larger datasets drove algorithmic innovations. Better algorithms demanded more sophisticated computing systems. This virtuous cycle continues to drive progress in the field today.

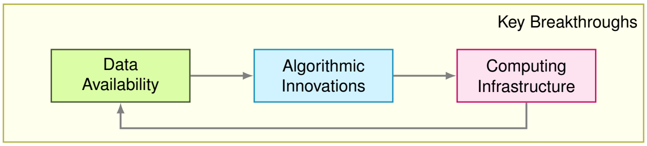


Figure 3.10: The virtuous cycle enabled by key breakthroughs in each layer.

The data revolution transformed what was possible with neural networks. The rise of the internet and digital devices created unprecedented access to training data. Image sharing platforms provided millions of labeled images. Digital text collections enabled language processing at scale. Sensor networks and IoT devices generated continuous streams of real-world data. This abundance of data provided the raw material needed for neural networks to learn complex patterns effectively.

Algorithmic innovations made it possible to harness this data effectively. New methods for initializing networks and controlling learning rates made training more stable. Techniques for preventing overfitting allowed models to generalize better to new data. Most importantly, researchers discovered that neural network performance scaled predictably with model size, computation, and data quantity, leading to increasingly ambitious architectures.

Computing infrastructure evolved to meet these growing demands. On the hardware side, graphics processing units (GPUs) provided the parallel processing capabilities needed for efficient neural network computation. Specialized AI accelerators like TPUs (Jouppi, Young, et al. 2017a) pushed performance further. High-bandwidth memory systems and fast interconnects addressed data movement challenges. Equally important were software advances—frameworks and libraries that made it easier to build and train networks, distributed computing systems that enabled training at scale, and tools for optimizing model deployment.

3.4 Neural Network Fundamentals

We can now examine the fundamental building blocks that make machine learning systems work. While the field has grown tremendously in sophistication, all modern neural networks, ranging from simple classifiers to large language models, share a common architectural foundation built upon basic computational units and principles.

This foundation begins with understanding how individual artificial neurons process information, how they are organized into layers, and how these layers are connected to form complete networks. By starting with these fundamental concepts, we can progressively build up to understanding more complex architectures and their applications.

Neural networks have come a long way since their inception in the 1950s, when the perceptron was first introduced. After a period of decline in popularity due to computational and theoretical limitations, the field saw a resurgence in the 2000s, driven by advancements in hardware (e.g., GPUs) and innovations like deep learning. These breakthroughs have made it possible to train networks with millions of parameters, enabling applications once considered impossible.

3.4.1 Basic Architecture

The architecture of a neural network determines how information flows through the system, from input to output. While modern networks can be tremendously complex, they all build upon a few key organizational principles that we will explore in the following sections. Understanding these principles is essential for both implementing neural networks and appreciating how they achieve their remarkable capabilities.

3.4.1.1 Neurons and Activations

The Perceptron is the basic unit or node that forms the foundation for more complex structures. It functions by taking multiple inputs, each representing a feature of the object under analysis, such as the characteristics of a home for predicting its price or the attributes of a song to forecast its popularity in music streaming services. These inputs are denoted as $x_1, x_2, ..., x_n$. A perceptron can be configured to perform either regression or classification tasks. For regression, the actual numerical output $\hat{y}$ is used. For classification, the output depends on whether $\hat{y}$ crosses a certain threshold. If $\hat{y}$ exceeds this threshold, the perceptron might output one class (e.g., 'yes'), and if it does not, another class (e.g., 'no').

Figure 3.11 illustrates the fundamental building blocks of a perceptron, which serves as the foundation for more complex neural networks. A perceptron can be thought of as a miniature decision-maker, utilizing its weights, bias, and

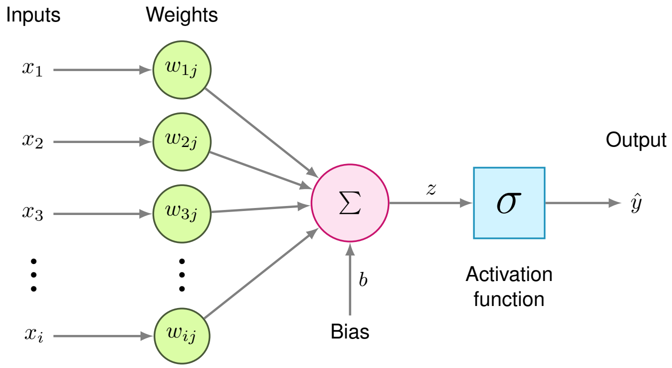


Figure 3.11: Perceptron. Conceived in the 1950s, perceptrons paved the way for developing more intricate neural networks and have been a fundamental building block in deep learning.

activation function to process inputs and generate outputs based on learned parameters. This concept forms the basis for understanding more intricate neural network architectures, such as multilayer perceptrons.

In these advanced structures, layers of perceptrons work in concert, with each layer's output serving as the input for the subsequent layer. This hierarchical arrangement creates a deep learning model capable of comprehending and modeling complex, abstract patterns within data. By stacking these simple units, neural networks gain the ability to tackle increasingly sophisticated tasks, from image recognition to natural language processing.

Each input x_i has a corresponding weight w_{ij}, and the perceptron simply multiplies each input by its matching weight. This operation is similar to linear regression, where the intermediate output, z, is computed as the sum of the products of inputs and their weights:

$$z = \sum (x_i \cdot w_{ij})$$

To this intermediate calculation, a bias term b is added, allowing the model to better fit the data by shifting the linear output function up or down. Thus, the intermediate linear combination computed by the perceptron including the bias becomes:

$$z = \sum (x_i \cdot w_{ij}) + b$$

Common activation functions include:[8]

- **ReLU (Rectified Linear Unit)**: Defined as $f(x) = \max(0, x)$, it introduces sparsity and accelerates convergence in deep networks. Its simplicity and effectiveness have made it the default choice in many modern architectures.
- **Sigmoid**: Historically popular, the sigmoid function maps inputs to a range between 0 and 1 but is prone to vanishing gradients in deeper architectures. It's particularly useful in binary classification problems where probabilities are needed.
- **Tanh**: Similar to sigmoid but maps inputs to a range of -1 to 1, centering the data. This centered output often leads to faster convergence in practice compared to sigmoid.

[8] Activation Function: A mathematical 'gate' in between the input from the previous layer and the output of the current layer, adding non-linearity to model complex patterns.

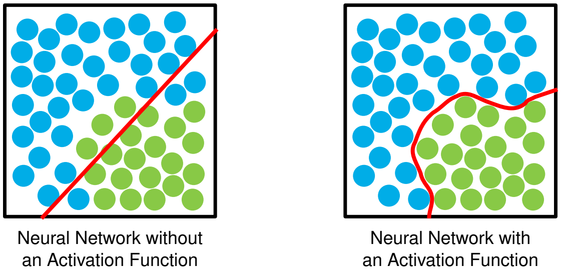


Figure 3.12: Activation functions enable the modeling of complex non-linear relationships. Source: Medium, Sachin Kaushik.

These activation functions transform the linear input sum into a non-linear output:

$$\hat{y} = \sigma(z)$$

Thus, the final output of the perceptron, including the activation function, can be expressed as:

Figure 3.12 shows an example where data exhibit a nonlinear pattern that could not be adequately modeled with a linear approach. The activation function enables the network to learn and represent complex relationships in the data, making it possible to solve sophisticated tasks like image recognition or speech processing.

Thus, the final output of the perceptron, including the activation function, can be expressed as:

$$z = \sigma\left(\sum (x_i \cdot w_{ij}) + b\right)$$

3.4.1.2 Layers and Connections

While a single perceptron can model simple decisions, the power of neural networks comes from combining multiple neurons into layers. A layer is a collection of neurons that process information in parallel. Each neuron in a layer operates independently on the same input but with its own set of weights and bias, allowing the layer to learn different features or patterns from the same input data.

In a typical neural network, we organize these layers hierarchically:

1. **Input Layer**: Receives the raw data features
2. **Hidden Layers**: Process and transform the data through multiple stages
3. **Output Layer**: Produces the final prediction or decision

Figure 3.13 illustrates this layered architecture. When data flows through these layers, each successive layer transforms the representation of the data, gradually building more complex and abstract features. This hierarchical processing is what gives deep neural networks their remarkable ability to learn complex patterns.

3.4.1.3 Data Flow and Transformations

As data flows through the network, it is transformed at each layer (l) to extract meaningful patterns. Each layer combines the input data using learned weights

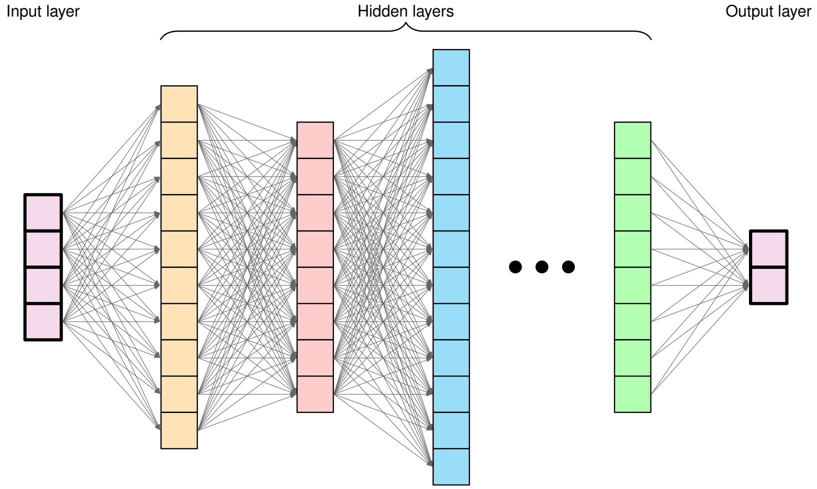


Figure 3.13: Neural network layers. Source: BrunelloN

and biases, then applies an activation function to introduce non-linearity. This process can be written mathematically as:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}$$

Where:

- $\mathbf{x}^{(l-1)}$ is the input vector from the previous layer
- $\mathbf{W}^{(l)}$ is the weight matrix for the current layer
- $\mathbf{b}^{(l)}$ is the bias vector
- $\mathbf{z}^{(l)}$ is the pre-activation output[9]

Now that we have covered the basics, Video 1 provides a great overview of how neural networks work using handwritten digit recognition. It introduces some new concepts that we will explore in more depth soon, but it serves as an excellent introduction.

! Important 1: Neural Network

Watch on YouTube

Watch on YouTube
Neural Network

Scan with your phone
to watch the video

3.4.2 Weights and Biases

3.4.2.1 Weight Matrices

Weights in neural networks determine how strongly inputs influence the output of a neuron. While we first discussed weights for a single perceptron, in larger networks, weights are organized into matrices for efficient computation across entire layers. For example, in a layer with n input features and m neurons, the weights form a matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$. Each column in this matrix represents the weights for a single neuron in the layer. This organization allows the network

[9] Pre-activation output: The output produced by a neuron in a neural network before the activation function is applied.

to process multiple inputs simultaneously, an essential feature for handling real-world data efficiently.

Let's consider how this extends our previous perceptron equations to handle multiple neurons simultaneously. For a layer of m neurons, instead of computing each neuron's output separately:

$$z_j = \sum_{i=1}^{n} (x_i \cdot w_{ij}) + b_j$$

We can compute all outputs at once using matrix multiplication:

$$\mathbf{z} = \mathbf{x}^T \mathbf{W} + \mathbf{b}$$

This matrix organization is more than just mathematical convenience; it reflects how modern neural networks are implemented for efficiency. Each weight w_{ij} represents the strength of the connection between input feature i and neuron j in the layer.

3.4.2.2 Connection Patterns

In the simplest and most common case, each neuron in a layer is connected to every neuron in the previous layer, forming what we call a "dense" or "fully-connected" layer. This pattern means that each neuron has the opportunity to learn from all available features from the previous layer.

Figure 3.14 illustrates these dense connections between layers. For a network with layers of sizes (n_1, n_2, n_3), the weight matrices would have dimensions:

- Between first and second layer: $\mathbf{W}^{(1)} \in \mathbb{R}^{n_1 \times n_2}$
- Between second and third layer: $\mathbf{W}^{(2)} \in \mathbb{R}^{n_2 \times n_3}$

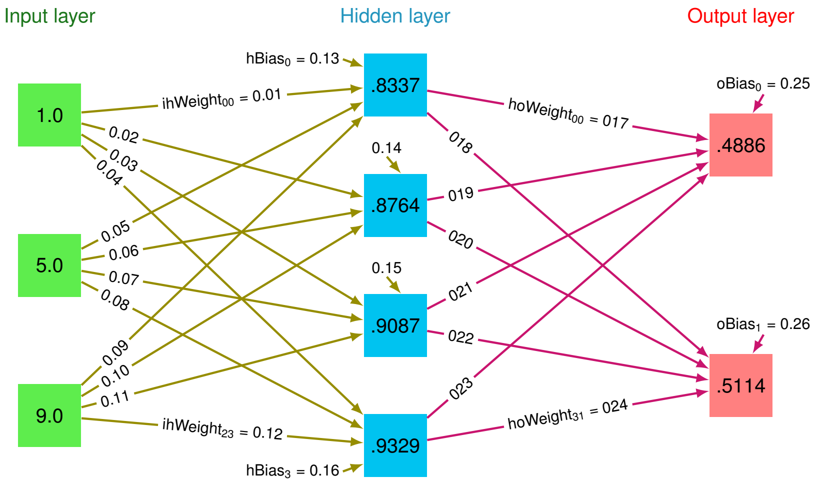


Figure 3.14: Dense connections between layers in a MLP. Source: J. McCaffrey

3.4.2.3 Bias Terms

Each neuron in a layer also has an associated bias term. While weights determine the relative importance of inputs, biases allow neurons to shift their activation functions. This shifting is crucial for learning, as it gives the network flexibility to fit more complex patterns.

For a layer with m neurons, the bias terms form a vector $\mathbf{b} \in \mathbb{R}^m$. When we compute the layer's output, this bias vector is added to the weighted sum of inputs:

$$\mathbf{z} = \mathbf{x}^T\mathbf{W} + \mathbf{b}$$

The bias terms effectively allow each neuron to have a different "threshold" for activation, making the network more expressive.

3.4.2.4 Parameter Organization

The organization of weights and biases across a neural network follows a systematic pattern. For a network with L layers, we maintain:

- A weight matrix $\mathbf{W}^{(l)}$ for each layer l
- A bias vector $\mathbf{b}^{(l)}$ for each layer l
- Activation functions $f^{(l)}$ for each layer l

This gives us the complete layer computation:

$$\mathbf{h}^{(l)} = f^{(l)}(\mathbf{z}^{(l)}) = f^{(l)}(\mathbf{h}^{(l-1)T}\mathbf{W}^{(l)} + \mathbf{b}^{(l)})$$

Where $\mathbf{h}^{(l)}$ represents the layer's output after applying the activation function.

3.4.3 Network Topology

Network topology describes how the basic building blocks we've discussed, such as neurons, layers, and connections, come together to form a complete neural network. We can best understand network topology through a concrete example. Consider the task of recognizing handwritten digits, a classic problem in deep learning using the MNIST[10] dataset.

3.4.3.1 Basic Structure

The fundamental structure of a neural network consists of three main components: input layer, hidden layers, and output layer. As shown in Figure 3.15a, a 28×28 pixel grayscale image of a handwritten digit must be processed through these layers to produce a classification output.

The input layer's width is directly determined by our data format. As shown in Figure 3.15b, for a 28×28 pixel image, each pixel becomes an input feature, requiring 784 input neurons $(28 \times 28 = 784)$. We can think of this either as a 2D grid of pixels or as a flattened vector of 784 values, where each value represents the intensity of one pixel.

The output layer's structure is determined by our task requirements. For digit classification, we use 10 output neurons, one for each possible digit (0-9). When presented with an image, the network produces a value for each output neuron,

[10] MNIST (Modified National Institute of Standards and Technology) is a large database of handwritten digits that has been widely used to train and test machine learning systems since its creation in 1998. The dataset consists of 60,000 training images and 10,000 testing images, each being a 28×28 pixel grayscale image of a single handwritten digit from 0 to 9.

where higher values indicate greater confidence that the image represents that particular digit.

Between these fixed input and output layers, we have flexibility in designing the hidden layer topology. The choice of hidden layer structure, including the number of layers to use and their respective widths, represents one of the fundamental design decisions in neural networks. Additional layers increase the network's depth, allowing it to learn more abstract features through successive transformations. The width of each layer provides capacity for learning different features at each level of abstraction.

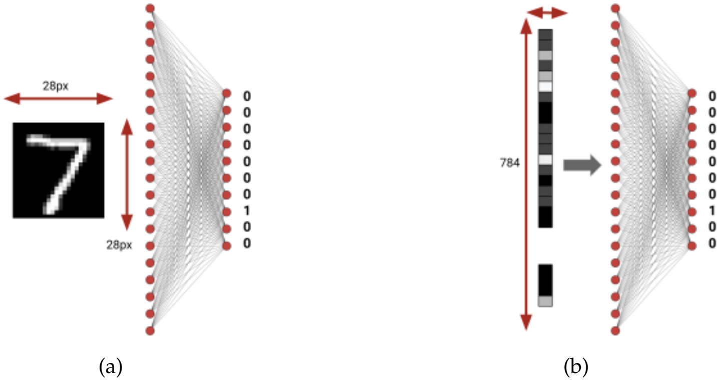


Figure 3.15: a) A neural network topology for classifying MNIST digits, showing how a 28×28 pixel image is processed. The image on the left shows the original digit, with dimensions labeled. The network on the right shows how each pixel connects to the hidden layers, ultimately producing 10 outputs for digit classification.

b) Alternative visualization of the MNIST network topology, showing how the 2D image is flattened into a 784-dimensional vector before being processed by the network. This representation emphasizes how spatial data is transformed into a format suitable for neural network processing.

These basic topological choices have significant implications for both the network's capabilities and its computational requirements. Each additional layer or neuron increases the number of parameters that must be stored and computed during both training and inference. However, without sufficient depth or width, the network may lack the capacity to learn complex patterns in the data.

3.4.3.2 Design Trade-offs

The design of neural network topology centers on three fundamental decisions: the number of layers (depth), the size of each layer (width), and how these layers connect. Each choice affects both the network's learning capability and its computational requirements.

Network depth determines the level of abstraction the network can achieve. Each layer transforms its input into a new representation, and stacking multiple layers allows the network to build increasingly complex features. In our MNIST example, a deeper network might first learn to detect edges, then combine these edges into strokes, and finally assemble strokes into complete digit patterns. However, adding layers isn't always beneficial—deeper networks increase computational cost substantially, can be harder to train due to vanishing gradients[11], and may require more sophisticated training techniques.

The width of each layer, which is determined by the number of neurons it contains, controls how much information the network can process in parallel at each stage. Wider layers can learn more features simultaneously but

[11] Vanishing Gradients: Problem in deep learning where gradient becomes so small that the model stops (or significantly slows down) learning.

require proportionally more parameters and computation. For instance, if a hidden layer is processing edge features in our digit recognition task, its width determines how many different edge patterns it can detect simultaneously.

A very important consideration in topology design is the total parameter count. For a network with layers of size $(n_1, n_2, \ldots, n_L)$, each pair of adjacent layers l and $l+1$ requires $n_l \times n_{l+1}$ weight parameters, plus n_{l+1} bias parameters. These parameters must be stored in memory and updated during training, making the parameter count a key constraint in practical applications.

When designing networks, we need to balance learning capacity, computational efficiency, and ease of training. While the basic approach connects every neuron to every neuron in the next layer (fully connected), this isn't always the most effective strategy. Sometimes, using fewer but more strategic connections, as seen in convolutional networks[12], can achieve better results with less computation. Consider our MNIST example—when humans recognize digits, we don't analyze every pixel independently but look for meaningful patterns like lines and curves. Similarly, we can design our network to focus on local patterns in the image rather than treating each pixel as completely independent.

Another important consideration is how information flows through the network. While the basic flow is from input to output, some network designs include additional paths for information to flow, such as skip connections or residual connections[13]. These alternative paths can make the network easier to train and more effective at learning complex patterns. Think of these as shortcuts that help information flow more directly when needed, similar to how our brain can combine both detailed and general impressions when recognizing objects.

These design decisions have significant practical implications for memory usage for storing network parameters, computational costs during both training and inference, training behavior and convergence, and the network's ability to generalize to new examples. The optimal balance of these trade-offs depends heavily on your specific problem, available computational resources, and dataset characteristics. Successful network design requires carefully weighing these factors against practical constraints.

3.4.3.3 Connection Patterns

Neural networks can be structured with different connection patterns between layers, each offering distinct advantages for learning and computation. Understanding these fundamental patterns provides insight into how networks process information and learn representations from data.

Dense connectivity represents the standard pattern where each neuron connects to every neuron in the subsequent layer. In our MNIST example, connecting our 784-dimensional input layer to a hidden layer of 100 neurons requires 78,400 weight parameters. This full connectivity enables the network to learn arbitrary relationships between inputs and outputs, but the number of parameters scales quadratically with layer width.

Sparse connectivity patterns introduce purposeful restrictions in how neurons connect between layers. Rather than maintaining all possible connections, neurons connect to only a subset of neurons in the adjacent layer. This approach

[12] Convolutional Networks (CNNs): A type of neural network architecture designed to process grid-structured input data, like images.

[13] Residual Connections (Skip Connections): Shortcut connections between layers in a neural network, helping mitigate the vanishing gradient problem by allowing gradients to flow directly through the network.

draws inspiration from biological neural systems, where neurons typically form connections with a limited number of other neurons. In visual processing tasks like our MNIST example, neurons might connect only to inputs representing nearby pixels, reflecting the local nature of visual features.

As networks grow deeper, the path from input to output becomes longer, potentially complicating the learning process. Skip connections address this by adding direct paths between non-adjacent layers. These connections provide alternative routes for information flow, supplementing the standard layer-by-layer progression. In our digit recognition example, skip connections might allow later layers to reference both high-level patterns and the original pixel values directly.

These connection patterns have significant implications for both the theoretical capabilities and practical implementation of neural networks. Dense connections maximize learning flexibility at the cost of computational efficiency. Sparse connections can reduce computational requirements while potentially improving the network's ability to learn structured patterns. Skip connections help maintain effective information flow in deeper networks.

3.4.3.4 Parameter Considerations

The arrangement of parameters (weights and biases) in a neural network determines both its learning capacity and computational requirements. While topology defines the network's structure, the initialization and organization of parameters plays a crucial role in learning and performance.

Parameter count grows with network width and depth. For our MNIST example, consider a network with a 784-dimensional input layer, two hidden layers of 100 neurons each, and a 10-neuron output layer. The first layer requires 78,400 weights and 100 biases, the second layer 10,000 weights and 100 biases, and the output layer 1,000 weights and 10 biases, totaling 89,610 parameters. Each must be stored in memory and updated during learning.

Parameter initialization is fundamental to network behavior. Setting all parameters to zero would cause neurons in a layer to behave identically, preventing diverse feature learning. Instead, weights are typically initialized randomly, while biases often start at small constant values or even zeros. The scale of these initial values matters significantly, as values that are too large or too small can lead to poor learning dynamics.

The distribution of parameters affects information flow through layers. In digit recognition, if weights are too small, important input details might not propagate to later layers. If too large, the network might amplify noise. Biases help adjust the activation threshold of each neuron, enabling the network to learn optimal decision boundaries.

Different architectures may impose specific constraints on parameter organization. Some share weights across network regions to encode position-invariant pattern recognition. Others might restrict certain weights to zero, implementing sparse connectivity patterns[14].

[14] Sparsity: In data structures, sparsity refers to elements being zero or absent. In neural networks, it can refer to the absence of connections between nodes.

3.5 Learning Process

Neural networks learn to perform tasks through a process of training on examples. This process transforms the network from its initial state, where its weights are randomly initialized, to a trained state where the weights encode meaningful patterns from the training data. Understanding this process is fundamental to both the theoretical foundations and practical implementations of deep learning systems.

3.5.1 Training Overview

The core principle of neural network training is supervised learning from labeled examples. Consider our MNIST digit recognition task: we have a dataset of 60,000 training images, each a 28×28 pixel grayscale image paired with its correct digit label. The network must learn the relationship between these images and their corresponding digits through an iterative process of prediction and weight adjustment.

Training operates as a loop, where each iteration involves processing a subset of training examples called a batch. For each batch, the network performs several key operations:

- Forward computation through the network layers to generate predictions
- Evaluation of prediction accuracy using a loss function[15]
- Computation of weight adjustments based on prediction errors
- Update of network weights to improve future predictions

This process can be expressed mathematically. Given an input image x and its true label y, the network computes its prediction:

$$\hat{y} = f(x; \theta)$$

where f represents the neural network function and θ represents all trainable parameters (weights and biases, which we discussed earlier). The network's error is measured by a loss function L:

$$\text{loss} = L(\hat{y}, y)$$

This error measurement drives the adjustment of network parameters through a process called "backpropagation," which we will examine in detail later.

In practice, training operates on batches of examples rather than individual inputs. For the MNIST dataset, each training iteration might process, for example, 32, 64, or 128 images simultaneously. This batch processing serves two purposes: it enables efficient use of modern computing hardware through parallel processing, and it provides more stable parameter updates by averaging errors across multiple examples.

The training cycle continues until the network achieves sufficient accuracy or reaches a predetermined number of iterations. Throughout this process, the loss function serves as a guide, with its minimization indicating improved network performance.

[15] Loss function: A method for evaluating how well the algorithm models the given training data. The lower the value, the better the model.

3.5.2 Forward Propagation

Forward propagation, as illustrated in Figure 3.16, is the core computational process in a neural network, where input data flows through the network's layers to generate predictions. Understanding this process is essential as it forms the foundation for both network inference and training. Let's examine how forward propagation works using our MNIST digit recognition example.

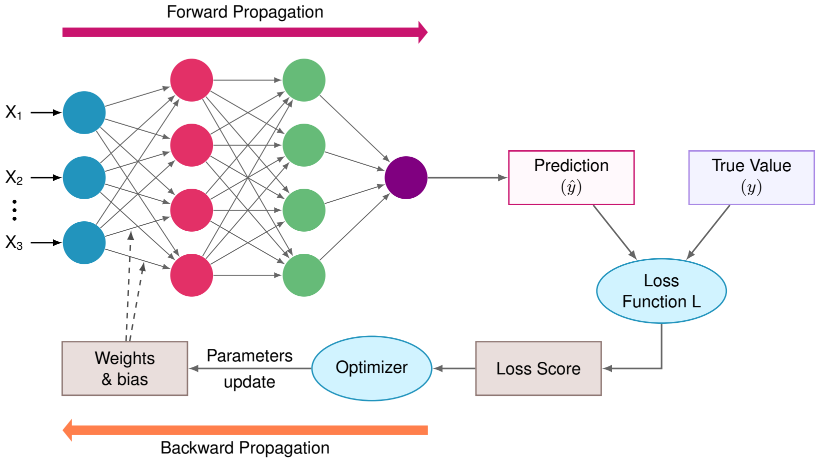


Figure 3.16: Neural networks, forward and backward propagation.

When an image of a handwritten digit enters our network, it undergoes a series of transformations through the layers. Each transformation combines the weighted inputs with learned patterns to progressively extract relevant features. In our MNIST example, a 28×28 pixel image is processed through multiple layers to ultimately produce probabilities for each possible digit (0-9).

The process begins with the input layer, where each pixel's grayscale value becomes an input feature. For MNIST, this means 784 input values $(28 \times 28 = 784)$, each normalized between 0 and 1. These values then propagate forward through the hidden layers, where each neuron combines its inputs according to its learned weights and applies a nonlinear activation function.

3.5.2.1 Layer Computation

The forward computation through a neural network proceeds systematically, with each layer transforming its inputs into increasingly abstract representations. In our MNIST network, this transformation process occurs in distinct stages.

At each layer, the computation involves two key steps: a linear transformation of inputs followed by a nonlinear activation. The linear transformation combines all inputs to a neuron using learned weights and a bias term. For a single neuron receiving inputs from the previous layer, this computation takes the form:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

where w_i represents the weights, x_i the inputs, and b the bias term. For an entire layer of neurons, we can express this more efficiently using matrix operations:

$$\mathbf{Z}^{(l)} = \mathbf{W}^{(l)}\mathbf{A}^{(l-1)} + \mathbf{b}^{(l)}$$

Here, $\mathbf{W}^{(l)}$ represents the weight matrix for layer l, $\mathbf{A}^{(l-1)}$ contains the activations from the previous layer, and $\mathbf{b}^{(l)}$ is the bias vector.

Following this linear transformation, each layer applies a nonlinear activation function f:

$$\mathbf{A}^{(l)} = f(\mathbf{Z}^{(l)})$$

This process repeats at each layer, creating a chain of transformations:

Input → Linear Transform → Activation → Linear Transform → Activation → … → Output

In our MNIST example, the pixel values first undergo a transformation by the first hidden layer's weights, converting the 784-dimensional input into an intermediate representation. Each subsequent layer further transforms this representation, ultimately producing a 10-dimensional output vector representing the network's confidence in each possible digit.

3.5.2.2 Mathematical Representation

The complete forward propagation process can be expressed as a composition of functions, each representing a layer's transformation. Let us formalize this mathematically, building on our MNIST example.

For a network with L layers, we can express the full forward computation as:

$$\mathbf{A}^{(L)} = f^{(L)}\Big(\mathbf{W}^{(L)}f^{(L-1)}\Big(\mathbf{W}^{(L-1)}\cdots\big(f^{(1)}(\mathbf{W}^{(1)}\mathbf{X}+\mathbf{b}^{(1)})\big)\cdots+\mathbf{b}^{(L-1)}\Big)+\mathbf{b}^{(L)}\Big)$$

While this nested expression captures the complete process, we typically compute it step by step:

1. First layer:

$$\mathbf{Z}^{(1)} = \mathbf{W}^{(1)}\mathbf{X} + \mathbf{b}^{(1)}$$
$$\mathbf{A}^{(1)} = f^{(1)}(\mathbf{Z}^{(1)})$$

2. Hidden layers $(l = 2, \ldots, L-1)$:

$$\mathbf{Z}^{(l)} = \mathbf{W}^{(l)}\mathbf{A}^{(l-1)} + \mathbf{b}^{(l)}$$
$$\mathbf{A}^{(l)} = f^{(l)}(\mathbf{Z}^{(l)})$$

3. Output layer:

$$\mathbf{Z}^{(L)} = \mathbf{W}^{(L)}\mathbf{A}^{(L-1)} + \mathbf{b}^{(L)}$$
$$\mathbf{A}^{(L)} = f^{(L)}(\mathbf{Z}^{(L)})$$

In our MNIST example, if we have a batch of B images, the dimensions of these operations are:

- Input $\mathbf{X}$: $B \times 784$
- First layer weights $\mathbf{W}^{(1)}$: $n_1 \times 784$
- Hidden layer weights $\mathbf{W}^{(l)}$: $n_l \times n_{l-1}$
- Output layer weights $\mathbf{W}^{(L)}$: $n_{L-1} \times 10$

3.5.2.3 Computational Process

To understand how these mathematical operations translate into actual computation, let's walk through the forward propagation process for a batch of MNIST images. This process illustrates how data is transformed from raw pixel values to digit predictions.

Consider a batch of 32 images entering our network. Each image starts as a 28×28 grid of pixel values, which we flatten into a 784-dimensional vector. For the entire batch, this gives us an input matrix $\mathbf{X}$ of size 32×784, where each row represents one image. The values are typically normalized to lie between 0 and 1.

The transformation at each layer proceeds as follows:

- **Input Layer Processing**: The network takes our input matrix $\mathbf{X}$ (32×784) and transforms it using the first layer's weights. If our first hidden layer has 128 neurons, $\mathbf{W}^{(1)}$ is a 784×128 matrix. The resulting computation $\mathbf{X}\mathbf{W}^{(1)}$ produces a 32×128 matrix.
- **Hidden Layer Transformations**: Each element in this matrix then has its corresponding bias added and passes through an activation function. For example, with a ReLU activation, any negative values become zero while positive values remain unchanged. This nonlinear transformation enables the network to learn complex patterns in the data.
- **Output Generation**: The final layer transforms its inputs into a 32×10 matrix, where each row contains 10 values corresponding to the network's confidence scores for each possible digit. Often, these scores are converted to probabilities using a softmax function:

$$P(\text{digit } j) = \frac{e^{z_j}}{\sum_{k=1}^{10} e^{z_k}}$$

For each image in our batch, this gives us a probability distribution over the possible digits. The digit with the highest probability becomes the network's prediction.

3.5.2.4 Practical Considerations

The implementation of forward propagation requires careful attention to several practical aspects that affect both computational efficiency and memory usage. These considerations become particularly important when processing large batches of data or working with deep networks.

Memory management plays an important role during forward propagation. Each layer's activations must be stored for potential use in the backward pass during training. For our MNIST example with a batch size of 32, if we have three hidden layers of sizes 128, 256, and 128, the activation storage requirements are:

- First hidden layer: $32 \times 128 = 4,096$ values
- Second hidden layer: $32 \times 256 = 8,192$ values
- Third hidden layer: $32 \times 128 = 4,096$ values
- Output layer: $32 \times 10 = 320$ values

This gives us a total of 16,704 values that must be maintained in memory for each batch during training. The memory requirements scale linearly with batch size and can become substantial for larger networks.

Batch processing introduces important trade-offs. Larger batches enable more efficient matrix operations and better hardware utilization but require more memory. For example, doubling the batch size to 64 would double our memory requirements for activations. This relationship between batch size, memory usage, and computational efficiency often guides the choice of batch size in practice.

The organization of computations also affects performance. Matrix operations can be optimized through careful memory layout and the use of specialized libraries. The choice of activation functions impacts not only the network's learning capabilities but also its computational efficiency, as some functions (like ReLU) are less expensive to compute than others (like tanh or sigmoid).

These considerations form the foundation for understanding the system requirements of neural networks, which we will explore in more detail in later chapters.

3.5.3 Loss Functions

Neural networks learn by measuring and minimizing their prediction errors. Loss functions provide the Algorithmic Structure for quantifying these errors, serving as the essential feedback mechanism that guides the learning process. Through loss functions, we can convert the abstract goal of "making good predictions" into a concrete optimization problem.

To understand the role of loss functions, let's continue with our MNIST digit recognition example. When the network processes a handwritten digit image, it outputs ten numbers representing its confidence in each possible digit (0-9). The loss function measures how far these predictions deviate from the true answer. For instance, if an image shows a "7", we want high confidence for digit "7" and low confidence for all other digits. The loss function penalizes the network when its prediction differs from this ideal.

Consider a concrete example: if the network sees an image of "7" and outputs confidences:

```
[0.1, 0.1, 0.1, 0.0, 0.0, 0.0, 0.2, 0.3, 0.1, 0.1]
```

The highest confidence (0.3) is assigned to digit "7", but this confidence is quite low, indicating uncertainty in the prediction. A good loss function would produce a high loss value here, signaling that the network needs significant improvement. Conversely, if the network outputs:

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.9, 0.0, 0.1]
```

The loss function should produce a lower value, as this prediction is much closer to ideal.

3.5.3.1 Basic Concepts

A loss function measures how far the network's predictions are from the correct answers. This difference is expressed as a single number: a lower loss means the predictions are more accurate, while a higher loss indicates the network needs improvement. During training, the loss function guides the network by helping it adjust its weights to make better predictions. For example, in recognizing handwritten digits, the loss will penalize predictions that assign low confidence to the correct digit.

Mathematically, a loss function L takes two inputs: the network's predictions $\hat{y}$ and the true values y. For a single training example in our MNIST task:

$$L(\hat{y}, y) = \text{measure of discrepancy between prediction and truth}$$

When training with batches of data, we typically compute the average loss across all examples in the batch:

$$L_{\text{batch}} = \frac{1}{B} \sum_{i=1}^{B} L(\hat{y}_i, y_i)$$

where B is the batch size and $(\hat{y}_i, y_i)$ represents the prediction and truth for the i-th example.

The choice of loss function depends on the type of task. For our MNIST classification problem, we need a loss function that can:

1. Handle probability distributions over multiple classes
2. Provide meaningful gradients for learning
3. Penalize wrong predictions effectively
4. Scale well with batch processing

3.5.3.2 Classification Losses

For classification tasks like MNIST digit recognition, "cross-entropy" loss[16] has emerged as the standard choice. This loss function is particularly well-suited for comparing predicted probability distributions with true class labels.

For a single digit image, our network outputs a probability distribution over the ten possible digits. We represent the true label as a one-hot vector where all entries are 0 except for a 1 at the correct digit's position. For instance, if the true digit is "7", the label would be:

$$y = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]$$

The cross-entropy loss for this example is:

$$L(\hat{y}, y) = -\sum_{j=1}^{10} y_j \log(\hat{y}_j)$$

where $\hat{y}_j$ represents the network's predicted probability for digit j. Given our one-hot encoding, this simplifies to:

$$L(\hat{y}, y) = -\log(\hat{y}_c)$$

[16] Cross-Entropy Loss: A type of loss function that measures the difference between two probability distributions.

where c is the index of the correct class. This means the loss depends only on the predicted probability for the correct digit—the network is penalized based on how confident it is in the right answer.

For example, if our network predicts the following probabilities for an image of "7":

```
Predicted: [0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.8, 0.0, 0.1]
True: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
```

The loss would be $-\log(0.8)$, which is approximately 0.223. If the network were more confident and predicted 0.9 for the correct digit, the loss would decrease to approximately 0.105.

3.5.3.3 Loss Computation

The practical computation of loss involves considerations for both numerical stability and batch processing. When working with batches of data, we compute the average loss across all examples in the batch.

For a batch of B examples, the cross-entropy loss becomes:

$$L_{\text{batch}} = -\frac{1}{B}\sum_{i=1}^{B}\sum_{j=1}^{10} y_{ij}\log(\hat{y}_{ij})$$

Computing this loss efficiently requires careful consideration of numerical precision. Taking the logarithm of very small probabilities can lead to numerical instability. Consider a case where our network predicts a probability of 0.0001 for the correct class. Computing $\log(0.0001)$ directly might cause underflow or result in imprecise values.

To address this, we typically implement the loss computation with two key modifications:

1. Add a small epsilon to prevent taking log of zero:

$$L = -\log(\hat{y} + \epsilon)$$

2. Apply the log-sum-exp trick[17] for numerical stability:

$$\text{softmax}(z_i) = \frac{\exp\left(z_i - \max(z)\right)}{\sum_j \exp\left(z_j - \max(z)\right)}$$

For our MNIST example with a batch size of 32, this means:

- Processing 32 sets of 10 probabilities
- Computing 32 individual loss values
- Averaging these values to produce the final batch loss

[17] Log-Sum-Exp trick: A method used in machine learning to prevent numerical underflow and overflow by normalizing the inputs of exponentiated operations.

3.5.3.4 Training Implications

Understanding how loss functions influence training helps explain key implementation decisions in deep learning systems.

During each training iteration, the loss value serves multiple purposes:

1. Performance Metric: It quantifies current network accuracy
2. Optimization Target: Its gradients guide weight updates
3. Convergence Signal: Its trend indicates training progress

For our MNIST classifier, monitoring the loss during training reveals the network's learning trajectory. A typical pattern might show:

- Initial high loss (~ 2.3, equivalent to random guessing among 10 classes)
- Rapid decrease in early training iterations
- Gradual improvement as the network fine-tunes its predictions
- Eventually stabilizing at a lower loss (~ 0.1, indicating confident correct predictions)

The loss function's gradients with respect to the network's outputs provide the initial error signal that drives backpropagation. For cross-entropy loss, these gradients have a particularly simple form: the difference between predicted and true probabilities. This mathematical property makes cross-entropy loss especially suitable for classification tasks, as it provides strong gradients even when predictions are very wrong.

The choice of loss function also influences other training decisions:

- Learning rate selection (larger loss gradients might require smaller learning rates)
- Batch size (loss averaging across batches affects gradient stability)
- Optimization algorithm behavior
- Convergence criteria

3.5.4 Backward Propagation

Backward propagation, often called backpropagation, is the algorithmic cornerstone of neural network training. While forward propagation computes predictions, backward propagation determines how to adjust the network's weights to improve these predictions. This process enables neural networks to learn from their mistakes.

To understand backward propagation, let's continue with our MNIST example. When the network predicts a "3" for an image of "7", we need a systematic way to adjust weights throughout the network to make this mistake less likely in the future. Backward propagation provides this by calculating how each weight contributed to the error.

The process begins at the network's output, where we compare the predicted digit probabilities with the true label. This error then flows backward through the network, with each layer's weights receiving an update signal based on their contribution to the final prediction. The computation follows the chain rule of calculus[18], breaking down the complex relationship between weights and final error into manageable steps.

[18] Chain rule of calculus: A basic theorem in calculus stating that the derivative of a composite function is the product of the derivative of the outer function and the derivative of the inner function.

Video 2 and Video 3 give a good high level overview of cost functions help neural networks learn

! Important 2: Gradient descent – Part 1

🖥 Watch on YouTube

! Important 3: Gradient descent – Part 2

🖥 Watch on YouTube

Watch on YouTube
Gradient descent – Part 1

Gradient descent – Part 2

Scan with your phone
to watch the video

3.5.4.1 Gradient Flow

The flow of gradients through a neural network follows a path opposite to the forward propagation. Starting from the loss at the output layer, gradients propagate backwards, computing how each layer, and ultimately each weight, influenced the final prediction error.

In our MNIST example, consider what happens when the network misclassifies a "7" as a "3". The loss function generates an initial error signal at the output layer, essentially indicating that the probability for "7" should increase while the probability for "3" should decrease. This error signal then propagates backward through the network layers.

For a network with L layers, the gradient flow can be expressed mathematically. At each layer l, we compute how the layer's output affected the final loss:

$$\frac{\partial L}{\partial \mathbf{A}^{(l)}} = \frac{\partial L}{\partial \mathbf{A}^{(l+1)}} \frac{\partial \mathbf{A}^{(l+1)}}{\partial \mathbf{A}^{(l)}}$$

This computation cascades backward through the network, with each layer's gradients depending on the gradients computed in the layer previous to it. The process reveals how each layer's transformation contributed to the final prediction error. For instance, if certain weights in an early layer strongly influenced a misclassification, they will receive larger gradient values, indicating a need for more substantial adjustment.

However, this process faces important challenges in deep networks. As gradients flow backward through many layers, they can either vanish or explode. When gradients are repeatedly multiplied through many layers, they can become exponentially small, particularly with sigmoid or tanh activation functions. This causes early layers to learn very slowly or not at all, as they receive negligible (vanishing) updates. Conversely, if gradient values are consistently greater than 1, they can grow exponentially, leading to unstable training and destructive weight updates.

3.5.4.2 Gradient Computation

The actual computation of gradients involves calculating several partial derivatives at each layer. For each layer, we need to determine how changes in weights, biases, and activations affect the final loss. These computations follow directly from the chain rule of calculus but must be implemented efficiently for practical neural network training.

At each layer l, we compute three main gradient components:

1. Weight Gradients:
$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \frac{\partial L}{\partial \mathbf{Z}^{(l)}} \mathbf{A}^{(l-1)^T}$$
2. Bias Gradients:
$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \frac{\partial L}{\partial \mathbf{Z}^{(l)}}$$
3. Input Gradients (for propagating to previous layer):
$$\frac{\partial L}{\partial \mathbf{A}^{(l-1)}} = \mathbf{W}^{(l)^T} \frac{\partial L}{\partial \mathbf{Z}^{(l)}}$$

In our MNIST example, consider the final layer where the network outputs digit probabilities. If the network predicted $[0.1, 0.2, 0.5, \ldots, 0.05]$ for an image of “7”, the gradient computation would:

1. Start with the error in these probabilities
2. Compute how weight adjustments would affect this error
3. Propagate these gradients backward to help adjust earlier layer weights

3.5.4.3 Implementation Aspects

The practical implementation of backward propagation requires careful consideration of computational resources and memory management. These implementation details significantly impact training efficiency and scalability.

Memory requirements during backward propagation stem from two main sources. First, we need to store the intermediate activations from the forward pass, as these are required for computing gradients. For our MNIST network with a batch size of 32, each layer’s activations must be maintained:

- Input layer: 32×784 values
- Hidden layers: $32 \times h$ values (where h is the layer width)
- Output layer: 32×10 values

Second, we need storage for the gradients themselves. For each layer, we must maintain gradients of similar dimensions to the weights and biases. Taking our previous example of a network with hidden layers of size 128, 256, and 128, this means storing:

- First layer gradients: 784×128 values
- Second layer gradients: 128×256 values
- Third layer gradients: 256×128 values
- Output layer gradients: 128×10 values

The computational pattern of backward propagation follows a specific sequence:

1. Compute gradients at current layer

2. Update stored gradients
3. Propagate error signal to previous layer
4. Repeat until input layer is reached

For batch processing, these computations are performed simultaneously across all examples in the batch, enabling efficient use of matrix operations and parallel processing capabilities.

3.5.5 Optimization Process

3.5.5.1 Gradient Descent Basics

The optimization process adjusts the network's weights to improve its predictions. Using a method called gradient descent, the network calculates how much each weight contributes to the error and updates it to reduce the loss. This process is repeated over many iterations, gradually refining the network's ability to make accurate predictions.

The fundamental update rule for gradient descent is:

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla_\theta L$$

where θ represents any network parameter (weights or biases), α is the learning rate, and $\nabla_\theta L$ is the gradient of the loss with respect to that parameter.

For our MNIST example, this means adjusting weights to improve digit classification accuracy. If the network frequently confuses "7"s with "1"s, gradient descent will modify the weights to better distinguish between these digits. The learning rate α controls how large these adjustments are—too large, and the network might overshoot optimal values; too small, and training will progress very slowly.

Video 4 demonstrates how the backpropagation math works in neural networks for those inclined towards a more theoretical foundation.

> **!** Important 4: Backpropagation
>
> 🖥 Watch on YouTube

Watch on YouTube
Backpropagation

Scan with your phone
to watch the video

3.5.5.2 Batch Processing

Neural networks typically process multiple examples simultaneously during training, an approach known as mini-batch gradient descent. Rather than updating weights after each individual image, we compute the average gradient over a batch of examples before performing the update.

For a batch of size B, the loss gradient becomes:

$$\nabla_\theta L_{\text{batch}} = \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta L_i$$

In our MNIST training, with a typical batch size of 32, this means:

1. Process 32 images through forward propagation

2. Compute loss for all 32 predictions
3. Average the gradients across all 32 examples
4. Update weights using this averaged gradient

3.5.5.3 Training Loop

The complete training process combines forward propagation, backward propagation, and weight updates into a systematic training loop. This loop repeats until the network achieves satisfactory performance or reaches a predetermined number of iterations.

A single pass through the entire training dataset is called an epoch. For MNIST, with 60,000 training images and a batch size of 32, each epoch consists of 1,875 batch iterations. The training loop structure is:

1. For each epoch:
    - Shuffle training data to prevent learning order-dependent patterns
    - For each batch:
        - Perform forward propagation
        - Compute loss
        - Execute backward propagation
        - Update weights using gradient descent
    - Evaluate network performance

During training, we monitor several key metrics:

- Training loss: average loss over recent batches
- Validation accuracy: performance on held-out test data
- Learning progress: how quickly the network improves

For our digit recognition task, we might observe the network's accuracy improve from 10% (random guessing) to over 95% through multiple epochs of training.

3.5.5.4 Practical Considerations

The successful implementation of neural network training requires attention to several key practical aspects that significantly impact learning effectiveness. These considerations bridge the gap between theoretical understanding and practical implementation.

Learning rate selection is perhaps the most critical parameter affecting training. For our MNIST network, the choice of learning rate dramatically influences the training dynamics. A large learning rate of 0.1 might cause unstable training where the loss oscillates or explodes as weight updates overshoot optimal values. Conversely, a very small learning rate of 0.0001 might result in extremely slow convergence, requiring many more epochs to achieve good performance. A moderate learning rate of 0.01 often provides a good balance between training speed and stability, allowing the network to make steady progress while maintaining stable learning.

Convergence monitoring provides crucial feedback during the training process. As training progresses, we typically observe the loss value stabilizing around a particular value, indicating the network is approaching a local optimum. The validation accuracy often plateaus as well, suggesting the network has extracted most of the learnable patterns from the data. The gap between training and validation performance offers insights into whether the network is overfitting or generalizing well to new examples.

Resource requirements become increasingly important as we scale neural network training. The memory footprint must accommodate both model parameters and the intermediate computations needed for backpropagation. Computation scales linearly with batch size, affecting training speed and hardware utilization. Modern training often leverages GPU acceleration, making efficient use of parallel computing capabilities crucial for practical implementation.

Training neural networks also presents several fundamental challenges. Overfitting occurs when the network becomes too specialized to the training data, performing well on seen examples but poorly on new ones. Gradient instability can manifest as either vanishing or exploding gradients, making learning difficult. The interplay between batch size, available memory, and computational resources often requires careful balancing to achieve efficient training while working within hardware constraints.

3.6 Prediction Phase

Neural networks serve two distinct purposes: learning from data during training and making predictions during inference. While we've explored how networks learn through forward propagation, backward propagation, and weight updates, the prediction phase operates differently. During inference, networks use their learned parameters to transform inputs into outputs without the need for learning mechanisms. This simpler computational process still requires careful consideration of how data flows through the network and how system resources are utilized. Understanding the prediction phase is crucial as it represents how neural networks are actually deployed to solve real-world problems, from classifying images to generating text predictions.

3.6.1 Inference Basics

3.6.1.1 Training vs Inference

The computation flow fundamentally changes when moving from training to inference. While training requires both forward and backward passes through the network to compute gradients and update weights, inference involves only the forward pass computation. This simpler flow means that each layer needs to perform only one set of operations, transforming inputs to outputs using the learned weights, rather than tracking intermediate values for gradient computation, as illustrated in Figure 3.17.

Parameter freezing is another another major distinction between training and inference phases. During training, weights and biases continuously update to minimize the loss function. In inference, these parameters remain fixed, acting as static transformations learned from the training data. This freezing

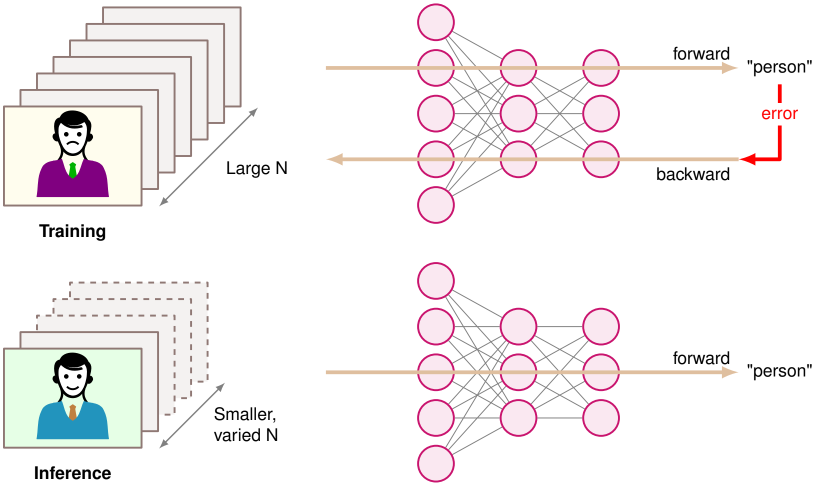


Figure 3.17: Comparing training versus inference data flow and computation.

of parameters not only simplifies computation but also enables optimizations impossible during training, such as weight quantization or pruning.

The structural difference between training loops and inference passes significantly impacts system design. Training operates in an iterative loop, processing multiple batches of data repeatedly across many epochs to refine the network's parameters. Inference, in contrast, typically processes each input just once, generating predictions in a single forward pass. This fundamental shift from iterative refinement to single-pass prediction influences how we architect systems for deployment.

Memory and computation requirements differ substantially between training and inference. Training demands considerable memory to store intermediate activations for backpropagation, gradients for weight updates, and optimization states. Inference eliminates these memory-intensive requirements, needing only enough memory to store the model parameters and compute a single forward pass. This reduction in memory footprint, coupled with simpler computation patterns, enables inference to run efficiently on a broader range of devices, from powerful servers to resource-constrained edge devices.

In general, the training phase requires more computational resources and memory for learning, while inference is streamlined for efficient prediction. Table 3.5 summarizes the key differences between training and inference.

Table 3.5: Key differences between training and inference phases in neural networks.

Aspect	Training	Inference
Computation Flow	Forward and backward passes, gradient computation	Forward pass only, direct input to output
Parameters	Continuously updated weights and biases	Fixed/frozen weights and biases
Processing Pattern	Iterative loops over multiple epochs	Single pass through the network
Memory Requirements	High – stores activations, gradients, optimizer state	Lower– stores only model | parameters and current input

Aspect	Training	Inference
Computational Needs	Heavy – gradient updates, backpropagation	Lighter – matrix multiplication only
Hardware Requirements	GPUs/specialized hardware for efficient training	Can run on simpler devices, including mobile/edge

This stark contrast between training and inference phases highlights why system architectures often differ significantly between development and deployment environments. While training requires substantial computational resources and specialized hardware, inference can be optimized for efficiency and deployed across a broader range of devices.

3.6.1.2 Basic Pipeline

The implementation of neural networks in practical applications requires a complete processing pipeline that extends beyond the network itself. This pipeline, which is illustrated in Figure 3.18 transforms raw inputs into meaningful outputs through a series of distinct stages, each essential for the system's operation. Understanding this complete pipeline provides critical insights into the design and deployment of machine learning systems.

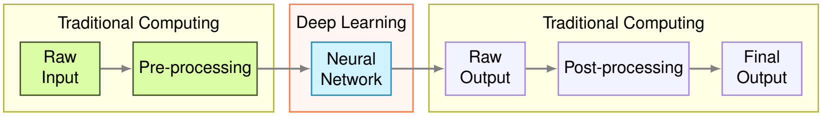


Figure 3.18: End-to-end workflow for the inference prediction phase.

The key thing to notice from the figure is that machine learning systems operate as hybrid architectures that combine conventional computing operations with neural network computations. The neural network component, focused on learned transformations through matrix operations, represents just one element within a broader computational framework. This framework encompasses both the preparation of input data and the interpretation of network outputs, processes that rely primarily on traditional computing methods.

Consider how data flows through the pipeline in Figure 3.18:

1. Raw inputs arrive in their original form, which might be images, text, sensor readings, or other data types
2. Pre-processing transforms these inputs into a format suitable for neural network consumption
3. The neural network performs its learned transformations
4. Raw outputs emerge from the network, often in numerical form
5. Post-processing converts these outputs into meaningful, actionable results

This pipeline structure reveals several fundamental characteristics of machine learning systems. The neural network, despite its computational sophistication, functions as a component within a larger system. Performance bottlenecks may arise at any stage of the pipeline, not exclusively within the neural network computation. System optimization must therefore consider the entire pipeline rather than focusing solely on the neural network's operation.

The hybrid nature of this architecture has significant implications for system implementation. While neural network computations may benefit from specialized hardware accelerators, pre- and post-processing operations typically execute on conventional processors. This distribution of computation across heterogeneous hardware resources represents a fundamental consideration in system design.

3.6.2 Pre-processing

The pre-processing stage transforms raw inputs into a format suitable for neural network computation. While often overlooked in theoretical discussions, this stage forms a critical bridge between real-world data and neural network operations. Consider our MNIST digit recognition example: before a handwritten digit image can be processed by the neural network we designed earlier, it must undergo several transformations. Raw images of handwritten digits arrive in various formats, sizes, and pixel value ranges. For instance, in Figure 3.19, we see that the digits are all of different sizes, and even the number 6 is written differently by the same person.

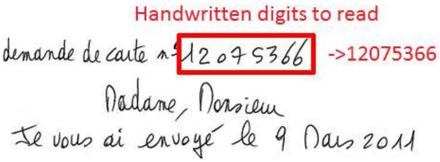

Handwritten digits to read
demande de carte n° 12075366
->12075366
Madame, Monsieur
Je vous ai envoyé le 9 Mars 2011

Figure 3.19: Images of handwritten digits. Source: O. Augereau

The pre-processing stage standardizes these inputs through conventional computing operations:

- Image scaling to the required 28×28 pixel dimensions, camera images are usually large(r).
- Pixel value normalization from $[0, 255]$ to $[0, 1]$, most cameras generate colored images.
- Flattening the 2D image array into a 784-dimensional vector, preparing it for the neural network.
- Basic validation to ensure data integrity, making sure the network predicted correctly.

What distinguishes pre-processing from neural network computation is its reliance on traditional computing operations rather than learned transformations. While the neural network learns to recognize digits through training, pre-processing operations remain fixed, deterministic transformations. This distinction has important system implications: pre-processing operates on conventional CPUs rather than specialized neural network hardware, and its performance characteristics follow traditional computing patterns.

The effectiveness of pre-processing directly impacts system performance. Poor normalization can lead to reduced accuracy, inconsistent scaling can introduce artifacts, and inefficient implementation can create bottlenecks. Understanding these implications helps in designing robust machine learning systems that perform well in real-world conditions.

3.6.3 Inference

The inference phase represents the operational state of a neural network, where learned parameters are used to transform inputs into predictions. Unlike the training phase we discussed earlier, inference focuses solely on forward computation with fixed parameters.

3.6.3.1 Network Initialization

Before processing any inputs, the neural network must be properly initialized for inference. This initialization phase involves loading the model parameters learned during training into memory. For our MNIST digit recognition network, this means loading specific weight matrices and bias vectors for each layer. Let's examine the exact memory requirements for our architecture:

- Input to first hidden layer:
  - Weight matrix: $784 \times 100 = 78,400$ parameters
  - Bias vector: 100 parameters
- First to second hidden layer:
  - Weight matrix: $100 \times 100 = 10,000$ parameters
  - Bias vector: 100 parameters
- Second hidden layer to output:
  - Weight matrix: $100 \times 10 = 1,000$ parameters
  - Bias vector: 10 parameters

In total, the network requires storage for 89,610 learned parameters (89,400 weights plus 210 biases). Beyond these fixed parameters, memory must also be allocated for intermediate activations during forward computation. For processing a single image, this means allocating space for:

- First hidden layer activations: 100 values
- Second hidden layer activations: 100 values
- Output layer activations: 10 values

This memory allocation pattern differs significantly from training, where additional memory was needed for gradients, optimizer states, and backpropagation computations.

3.6.3.2 Forward Pass Computation

During inference, data propagates through the network's layers using the initialized parameters. This forward propagation process, while similar in structure to its training counterpart, operates with different computational constraints and optimizations. The computation follows a deterministic path from input to output, transforming the data at each layer using learned parameters.

For our MNIST digit recognition network, consider the precise computations at each layer. The network processes a pre-processed image represented as a 784-dimensional vector through successive transformations:

1. First Hidden Layer Computation:

   - Input transformation: 784 inputs combine with 78,400 weights through matrix multiplication
   - Linear computation: $\mathbf{z}^{(1)} = \mathbf{x}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$
   - Activation: $\mathbf{a}^{(1)} = \text{ReLU}(\mathbf{z}^{(1)})$
   - Output: 100-dimensional activation vector

2. Second Hidden Layer Computation:
   - Input transformation: 100 values combine with 10,000 weights
   - Linear computation: $\mathbf{z}^{(2)} = \mathbf{a}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$
   - Activation: $\mathbf{a}^{(2)} = \text{ReLU}(\mathbf{z}^{(2)})$
   - Output: 100-dimensional activation vector

3. Output Layer Computation:
   - Final transformation: 100 values combine with 1,000 weights
   - Linear computation: $\mathbf{z}^{(3)} = \mathbf{a}^{(2)}\mathbf{W}^{(3)} + \mathbf{b}^{(3)}$
   - Activation: $\mathbf{a}^{(3)} = \text{softmax}(\mathbf{z}^{(3)})$
   - Output: 10 probability values

Table 3.6 shows how these computations, while mathematically identical to training-time forward propagation, show important operational differences:

Table 3.6: Operational characteristics of forward pass computation during training versus inference

Characteristic	Training Forward Pass	Inference Forward Pass
Activation Storage	Maintains complete activation history for backpropagation	Retains only current layer activations
Memory Pattern	Preserves intermediate states throughout forward pass	Releases memory after layer computation completes
Computational Flow	Structured for gradient computation preparation	Optimized for direct output generation
Resource Profile	Higher memory requirements for training operations	Minimized memory footprint for efficient execution

This streamlined computation pattern enables efficient inference while maintaining the network's learned capabilities. The reduction in memory requirements and simplified computational flow make inference particularly suitable for deployment in resource-constrained environments, such as Mobile ML and Tiny ML.

3.6.3.3 Resource Requirements

Neural networks consume computational resources differently during inference compared to training. During inference, resource utilization focuses primarily on efficient forward pass computation and minimal memory overhead. Let's examine the specific requirements for our MNIST digit recognition network.

Memory requirements during inference can be precisely quantified:

1. Static Memory (Model Parameters):
   - Layer 1: 78,400 weights + 100 biases

   - Layer 2: 10,000 weights + 100 biases
   - Layer 3: 1,000 weights + 10 biases
   - Total: 89,610 parameters (≈ 358.44 KB at 32-bit floating point precision)

2. Dynamic Memory (Activations):
   - Layer 1 output: 100 values
   - Layer 2 output: 100 values
   - Layer 3 output: 10 values
   - Total: 210 values (≈ 0.84 KB at 32-bit floating point precision)

Computational requirements follow a fixed pattern for each input:

- First layer: 78,400 multiply-adds
- Second layer: 10,000 multiply-adds
- Output layer: 1,000 multiply-adds
- Total: 89,400 multiply-add operations per inference

This resource profile stands in stark contrast to training requirements, where additional memory for gradients and computational overhead for backpropagation significantly increase resource demands. The predictable, streamlined nature of inference computations enables various optimization opportunities and efficient hardware utilization.

3.6.3.4 Optimization Opportunities

The fixed nature of inference computation presents several opportunities for optimization that are not available during training. Once a neural network's parameters are frozen, the predictable pattern of computation allows for systematic improvements in both memory usage and computational efficiency.

Batch size selection represents a fundamental trade-off in inference optimization. During training, large batches were necessary for stable gradient computation, but inference offers more flexibility. Processing single inputs minimizes latency, making it ideal for real-time applications where immediate responses are crucial. However, batch processing can significantly improve throughput by better utilizing parallel computing capabilities, particularly on GPUs. For our MNIST network, consider the memory implications: processing a single image requires storing 210 activation values, while a batch of 32 images requires 6,720 activation values but can process images up to 32 times faster on parallel hardware.

Memory management during inference can be significantly more efficient than during training. Since intermediate values are only needed for forward computation, memory buffers can be carefully managed and reused. The activation values from each layer need only exist until the next layer's computation is complete. This enables in-place operations where possible, reducing the total memory footprint. Furthermore, the fixed nature of inference allows for precise memory alignment and access patterns optimized for the underlying hardware architecture.

Hardware-specific optimizations become particularly important during inference. On CPUs, computations can be organized to maximize cache utilization and take advantage of SIMD (Single Instruction, Multiple Data) capabilities. GPU deployments benefit from optimized matrix multiplication routines and efficient memory transfer patterns. These optimizations extend beyond pure computational efficiency, as they can significantly impact power consumption and hardware utilization, critical factors in real-world deployments.

The predictable nature of inference also enables more aggressive optimizations like reduced numerical precision. While training typically requires 32-bit floating-point precision to maintain stable gradient computation, inference can often operate with 16-bit or even 8-bit precision while maintaining acceptable accuracy. For our MNIST network, this could reduce the memory footprint from 358.44 KB to 179.22 KB or even 89.61 KB, with corresponding improvements in computational efficiency.

These optimization principles, while illustrated through our simple MNIST feedforward network, represent only the foundation of neural network optimization. More sophisticated architectures introduce additional considerations and opportunities. Convolutional Neural Networks (CNNs), for instance, present unique optimization opportunities in handling spatial data and filter operations. Recurrent Neural Networks (RNNs) require careful consideration of sequential computation and state management. Transformer architectures introduce distinct patterns of attention computation and memory access. These architectural variations and their optimizations will be explored in detail in subsequent chapters, particularly when we discuss deep learning architectures, model optimizations, and efficient AI implementations.

3.6.4 Post-processing

The transformation of neural network outputs into actionable predictions requires a return to traditional computing paradigms. Just as pre-processing bridges real-world data to neural computation, post-processing bridges neural outputs back to conventional computing systems. This completes the hybrid computing pipeline we examined earlier, where neural and traditional computing operations work in concert to solve real-world problems.

The complexity of post-processing extends beyond simple mathematical transformations. Real-world systems must handle uncertainty, validate outputs, and integrate with larger computing systems. In our MNIST example, a digit recognition system might require not just the most likely digit, but also confidence measures to determine when human intervention is needed. This introduces additional computational steps: confidence thresholds, secondary prediction checks, and error handling logic, all of which are implemented in traditional computing frameworks.

The computational requirements of post-processing differ significantly from neural network inference. While inference benefits from parallel processing and specialized hardware, post-processing typically runs on conventional CPUs and follows sequential logic patterns. This return to traditional computing brings both advantages and constraints. Operations are more flexible and easier to modify than neural computations, but they may become bottlenecks if not carefully implemented. For instance, computing softmax probabilities[19] for a batch of predictions requires different optimization strategies than the matrix multiplications of neural network layers.

System integration considerations often dominate post-processing design. Output formats must match downstream system requirements, error handling must align with broader system protocols, and performance must meet system-level constraints. In a complete mail sorting system, the post-processing stage must not only identify digits but also format these predictions for the sorting machinery, handle uncertainty cases appropriately, and maintain processing speeds that match physical mail flow rates.

This return to traditional computing paradigms completes the hybrid nature of machine learning systems. Just as pre-processing prepared real-world data for neural computation, post-processing adapts neural outputs for real-world use. Understanding this hybrid nature, the interplay between neural and traditional computing, is essential for designing and implementing effective machine learning systems.

3.7 Case Study: USPS Postal Service

3.7.1 Real-world Problem

The United States Postal Service (USPS) processes over 100 million pieces of mail daily, each requiring accurate routing based on handwritten ZIP codes. In the early 1990s, this task was primarily performed by human operators, making it one of the largest manual data entry operations in the world. The automation of this process through neural networks represents one of the earliest and most successful large-scale deployments of artificial intelligence, embodying many of the principles we've explored in this chapter.

Consider the complexity of this task: a ZIP code recognition system must process images of handwritten digits captured under varying conditions, different writing styles, pen types, paper colors, and environmental factors (see Figure 3.20). It must make accurate predictions within milliseconds to maintain mail processing speeds. Furthermore, errors in recognition can lead to significant delays and costs from misrouted mail. This real-world constraint meant the system needed not just high accuracy, but also reliable measures of prediction confidence to identify when human intervention was necessary.

This challenging environment presented requirements spanning every aspect of neural network implementation we've discussed, from biological inspiration to practical deployment considerations. The success or failure of the system would depend not just on the neural network's accuracy, but on the entire pipeline from image capture through to final sorting decisions.

3.7.2 System Development

The development of the USPS digit recognition system required careful consideration at every stage, from data collection to deployment. This process illustrates how theoretical principles of neural networks translate into practical engineering decisions.

[19] Softmax: A function that converts logits into probabilities by scaling them based on a temperature parameter.

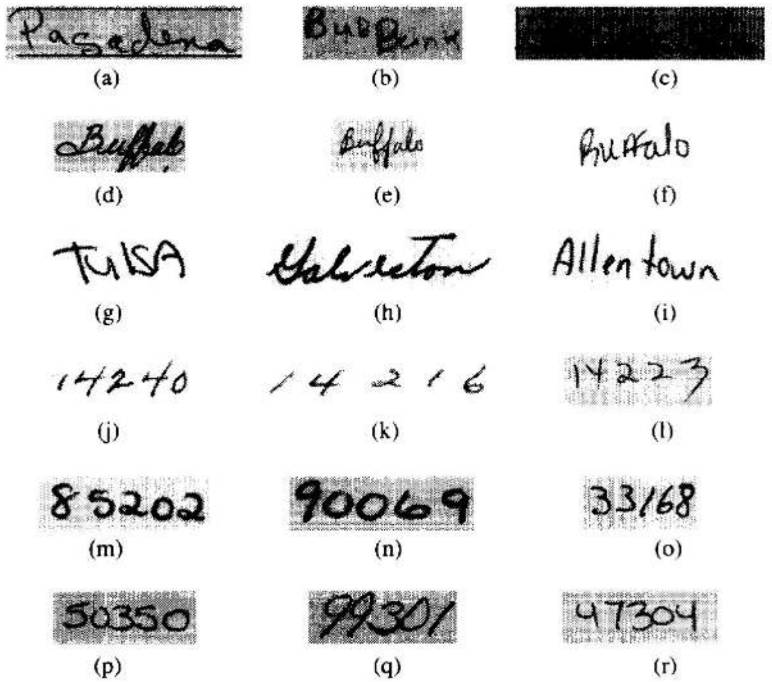


Figure 3.20: Example handwritten zipcodes from the USPS dataset.

Data collection presented the first major challenge. Unlike controlled laboratory environments, postal facilities needed to process mail pieces with tremendous variety. The training dataset had to capture this diversity. Digits written by people of different ages, educational backgrounds, and writing styles formed just part of the challenge. Envelopes came in varying colors and textures, and images were captured under different lighting conditions and orientations. This extensive data collection effort later contributed to the creation of the MNIST database we've used in our examples.

The network architecture design required balancing multiple constraints. While deeper networks might achieve higher accuracy, they would also increase processing time and computational requirements. Processing 28×28 pixel images of individual digits needed to complete within strict time constraints while running reliably on available hardware. The network had to maintain consistent accuracy across varying conditions, from well-written digits to hurried scrawls.

Training the network introduced additional complexity. The system needed to achieve high accuracy not just on a test dataset, but on the endless variety of real-world handwriting styles. Careful preprocessing normalized input images to account for variations in size and orientation. Data augmentation techniques increased the variety of training samples. The team validated performance across different demographic groups and tested under actual operating conditions to ensure robust performance.

The engineering team faced a critical decision regarding confidence thresholds[20]. Setting these thresholds too high would route too many pieces to human operators, defeating the purpose of automation. Setting them too low would risk delivery errors. The solution emerged from analyzing the confidence distributions of correct versus incorrect predictions. This analysis established thresholds that optimized the tradeoff between automation rate and error rate, ensuring efficient operation while maintaining acceptable accuracy.

3.7.3 Complete Pipeline

Following a single piece of mail through the USPS recognition system illustrates how the concepts we've discussed integrate into a complete solution. The journey from physical mail piece to sorted letter demonstrates the interplay between traditional computing, neural network inference, and physical machinery.

The process begins when an envelope reaches the imaging station. High-speed cameras capture the ZIP code region at rates exceeding several pieces of mail (e.g. 10) pieces per second. This image acquisition process must adapt to varying envelope colors, handwriting styles, and environmental conditions. The system must maintain consistent image quality despite the speed of operation, as motion blur and proper illumination present significant engineering challenges.

Pre-processing transforms these raw camera images into a format suitable for neural network analysis. The system must locate the ZIP code region, segment individual digits, and normalize each digit image. This stage employs traditional computer vision techniques: image thresholding[21] adapts to envelope background color, connected component analysis[22] identifies individual digits, and size normalization produces standard 28×28 pixel images. Speed remains critical; these operations must complete within milliseconds to maintain throughput.

The neural network then processes each normalized digit image. The trained network, with its 89,610 parameters (as we detailed earlier), performs forward propagation to generate predictions. Each digit passes through two hidden layers of 100 neurons each, ultimately producing ten output values representing digit probabilities. This inference process, while computationally intensive, benefits from the optimizations we discussed in the previous section.

Post-processing converts these neural network outputs into sorting decisions. The system applies confidence thresholds to each digit prediction. A complete ZIP code requires high confidence in all five digits, a single uncertain digit flags the entire piece for human review. When confidence meets thresholds, the system transmits sorting instructions to mechanical systems that physically direct the mail piece to its appropriate bin.

The entire pipeline operates under strict timing constraints. From image capture to sorting decision, processing must complete before the mail piece reaches its sorting point. The system maintains multiple pieces in various pipeline stages simultaneously, requiring careful synchronization between computing and mechanical systems. This real-time operation illustrates why the optimizations we discussed in inference and post-processing become crucial in practical applications.

[20] Confidence thresholds: Predetermined limits set to decide when the model's prediction is to be accepted. Influences system efficiency and accuracy.

[21] Image Thresholding: A technique in image processing that converts grayscale images into binary images.

[22] Connected Component Analysis: An operation in image processing used to distinguish and identify disjoint objects within an image.

3.7.4 Results and Impact

The implementation of neural network-based ZIP code recognition transformed USPS mail processing operations. By 2000, several facilities across the country utilized this technology, processing millions of mail pieces daily. This real-world deployment demonstrated both the potential and limitations of neural network systems in mission-critical applications.

Performance metrics revealed interesting patterns that validate many of the principles discussed earlier in this chapter. The system achieved its highest accuracy on clearly written digits, similar to those in the training data. However, performance varied significantly with real-world factors. Lighting conditions affected pre-processing effectiveness. Unusual writing styles occasionally confused the neural network. Environmental vibrations could also impact image quality. These challenges led to continuous refinements in both the physical system and the neural network pipeline.

The economic impact proved substantial. Prior to automation, manual sorting required operators to read and key in ZIP codes at an average rate of one piece per second. The neural network system processed pieces at ten times this rate while reducing labor costs and error rates. However, the system didn't eliminate human operators entirely; their role shifted to handling uncertain cases and maintaining system performance. This hybrid approach, combining artificial and human intelligence, became a model for other automation projects.

The system also revealed important lessons about deploying neural networks in production environments. Training data quality proved crucial; the network performed best on digit styles well-represented in its training set. Regular retraining helped adapt to evolving handwriting styles. Maintenance required both hardware specialists and machine learning experts, introducing new operational considerations. These insights influenced subsequent deployments of neural networks in other industrial applications.

Perhaps most importantly, this implementation demonstrated how theoretical principles translate into practical constraints. The biological inspiration of neural networks provided the foundation for digit recognition, but successful deployment required careful consideration of system-level factors: processing speed, error handling, maintenance requirements, and integration with existing infrastructure. These lessons continue to inform modern machine learning deployments, where similar challenges of scale, reliability, and integration persist.

3.7.5 Key Takeaways

The USPS ZIP code recognition system is an excellent example of the journey from biological inspiration to practical neural network deployment that we've explored throughout this chapter. It demonstrates how the basic principles of neural computation, from pre-processing through inference to post-processing, come together in solving real-world problems.

The system's development shows why understanding both the theoretical foundations and practical considerations is crucial. While the biological visual system processes handwritten digits effortlessly, translating this capability into an artificial system required careful consideration of network architecture, training procedures, and system integration.

The success of this early large-scale neural network deployment helped establish many practices we now consider standard: the importance of comprehensive training data, the need for confidence metrics, the role of pre- and post-processing, and the critical nature of system-level optimization.

As we move forward to explore more complex architectures and applications in subsequent chapters, this case study reminds us that successful deployment requires mastery of both fundamental principles and practical engineering considerations.

3.8 Conclusion

In this chapter, we explored the foundational concepts of neural networks, bridging the gap between biological inspiration and artificial implementation. We began by examining the remarkable efficiency and adaptability of the human brain, uncovering how its principles influence the design of artificial neurons. From there, we delved into the behavior of a single artificial neuron, breaking down its components and operations. This understanding laid the groundwork for constructing neural networks, where layers of interconnected neurons collaborate to tackle increasingly complex tasks.

The progression from single neurons to network-wide behavior underscored the power of hierarchical learning, where each layer extracts and transforms patterns from raw data into meaningful abstractions. We examined both the learning process and the prediction phase, showing how neural networks first refine their performance through training and then deploy that knowledge through inference. The distinction between these phases revealed important system-level considerations for practical implementations.

Our exploration of the complete processing pipeline, from pre-processing through inference to post-processing, highlighted the hybrid nature of machine learning systems, where traditional computing and neural computation work together. The USPS case study demonstrated how these theoretical principles translate into practical applications, revealing both the power and complexity of deployed neural networks. These real-world considerations, from data collection to system integration, form an essential part of understanding machine learning systems.

In the next chapter, we will expand on these ideas, exploring sophisticated deep learning architectures such as convolutional and recurrent neural networks. These architectures are tailored to process diverse data types, from images and text to time series, enabling breakthroughs across a wide range of applications. By building on the concepts introduced here, we will gain a deeper appreciation for the design, capabilities, and versatility of modern deep learning systems.

3.9 Resources

ℹ Slides

- Past, Present, and Future of ML.
- Thinking About Loss.
- Minimizing Loss.
- First Neural Network.
- Understanding Neurons.
- Intro to CLassification.
- Training, Validation, and Test Data.
- Intro to Convolutions.

! Videos

- Video 1
- Video 2
- Video 3
- Video 4

🔥 Exercises

Coming soon.

Chapter 4

DNN Architectures

Figure 4.1: *DALL·E 3 Prompt: A visually striking rectangular image illustrating the interplay between deep learning algorithms like CNNs, RNNs, and Attention Networks, interconnected with machine learning systems. The composition features neural network diagrams blending seamlessly with representations of computational systems such as processors, graphs, and data streams. Bright neon tones contrast against a dark futuristic background, symbolizing cutting-edge technology and intricate system complexity.*

Purpose

What recurring patterns emerge across modern deep learning architectures, and how do these patterns enable systematic approaches to AI system design?

Deep learning architectures represent a convergence of computational patterns that form the building blocks of modern AI systems. These foundational patterns, ranging from convolutional structures to attention mechanisms, reveal how complex models arise from simple, repeatable components. The examination of these architectural elements provides insights into the systematic construction of flexible, efficient AI systems, establishing core principles that influence every aspect of system design and deployment. These structural insights illuminate the path toward creating scalable, adaptable solutions across diverse application domains.

Learning Objectives

- Map fundamental neural network concepts to deep learning architectures (dense, spatial, temporal, attention-based).
- Analyze how architectural patterns shape computational and memory demands.
- Evaluate system-level impacts of architectural choices on system attributes.
- Compare architectures' hardware mapping and identify optimization strategies.
- Assess trade-offs between complexity and system needs for specific applications.

4.1 Overview

Deep learning architecture stands for specific representation or organizations of neural network components, the neurons, weights, and connections (as introduced in Chapter 3), arranged to efficiently process different types of patterns in data. While the previous chapter established the fundamental building blocks of neural networks, in this chapter we examine how these components are structured into architectures that map efficiently to computer systems.

Neural network architectures have evolved to address specific pattern processing challenges. Whether processing arbitrary feature relationships, exploiting spatial patterns, managing temporal dependencies, or handling dynamic information flow, each architectural pattern emerged from particular computational needs. These architectures, from a computer systems perspective, require an examination of how their computational patterns map to system resources.

Most often the architectures are discussed in terms of their algorithmic structures (MLPs, CNNs, RNNs, Transformers). However, in this chapter we take a more fundamental approach by examining how their computational patterns map to hardware resources. Each section analyzes how specific pattern processing needs influence algorithmic structure and how these structures map to computer system resources. The implications for computer system design require examining how their computational patterns map to hardware resources. The mapping from algorithmic requirements to computer system design involves several key considerations:

1. Memory access patterns: How data moves through the memory hierarchy
2. Computation characteristics: The nature and organization of arithmetic operations
3. Data movement: Requirements for on-chip and off-chip data transfer
4. Resource utilization: How computational and memory resources are allocated

For example, dense connectivity patterns generate different memory bandwidth demands than localized processing structures. Similarly, stateful processing creates distinct requirements for on-chip memory organization compared to stateless operations. Getting a firm grasp on these mappings is important for modern computer architects and system designers who must implement these algorithms efficiently in hardware.

4.2 Multi-Layer Perceptrons: Dense Pattern Processing

Multi-Layer Perceptrons (MLPs) represent the most direct extension of neural networks into deep architectures. Unlike more specialized networks, MLPs process each input element with equal importance, making them versatile but computationally intensive. Their architecture, while simple, establishes fundamental computational patterns that appear throughout deep learning systems. These patterns were initially formalized by the introduction of the Universal Approximation Theorem (UAT) (Cybenko 1992; Hornik, Stinchcombe, and White 1989), which states that a sufficiently large MLP with non-linear activation functions can approximate any continuous function on a compact domain, given suitable weights and biases.

When applied to the MNIST handwritten digit recognition challenge, an MLP reveals its computational power by transforming a complex 28×28 pixel image into a precise digit classification. By treating each of the 784 pixels as an equally weighted input, the network learns to decompose visual information through a systematic progression of layers, converting raw pixel intensities into increasingly abstract representations that capture the essential characteristics of handwritten digits.

4.2.1 Pattern Processing Needs

Deep learning systems frequently encounter problems where any input feature could potentially influence any output, as there are no inherent constraints on these relationships. Consider analyzing financial market data: any economic indicator might affect any market outcome or in natural language processing, where the meaning of a word could depend on any other word in the sentence. These scenarios demand an architectural pattern capable of learning arbitrary relationships across all input features.

Dense pattern processing addresses this fundamental need by enabling several key capabilities. First, it allows unrestricted feature interactions where each output can depend on any combination of inputs. Second, it facilitates learned feature importance, allowing the system to determine which connections matter rather than having them prescribed. Finally, it provides adaptive representation, enabling the network to reshape its internal representations based on the data.

For example, in the MNIST digit recognition task, while humans might focus on specific parts of digits (like loops in ‘6’ or crossings in ‘8’), we cannot definitively say which pixel combinations are important for classification. A ‘7’ written with a serif could share pixel patterns with a ‘2’, while variations in handwriting mean discriminative features might appear anywhere in the image. This uncertainty about feature relationships necessitates a dense processing approach where every pixel can potentially influence the classification decision.

4.2.2 Algorithmic Structure

To enable unrestricted feature interactions, MLPs implement a direct algorithmic solution: connect everything to everything. This is realized through a series of fully-connected layers, where each neuron connects to every neuron in adjacent layers. The dense connectivity pattern translates mathematically into matrix multiplication operations. As shown in Figure 4.2, each layer transforms its input through matrix multiplication followed by element-wise activation:

$$\mathbf{h}^{(l)} = f\big(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\big)$$

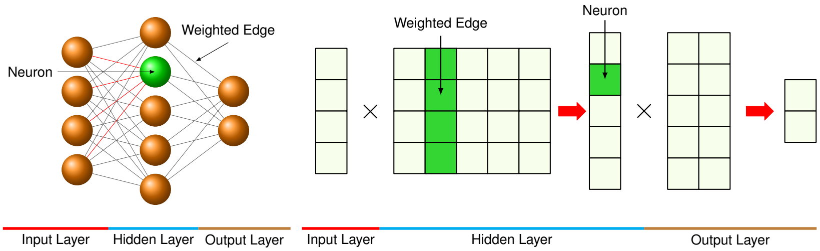


Figure 4.2: MLP layers and its associated matrix representation. Source: Reagen et al. (2017)

The dimensions of these operations reveal the computational scale of dense pattern processing:

- Input vector: $\mathbf{h}^{(0)} \in \mathbb{R}^{d_{\text{in}}}$ represents all potential input features
- Weight matrices: $\mathbf{W}^{(l)} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ capture all possible input-output relationships
- Output vector: $\mathbf{h}^{(l)} \in \mathbb{R}^{d_{\text{out}}}$ produces transformed representations

In the MNIST example, this means:

- Each 784-dimensional input (28×28 pixels) connects to every neuron in the first hidden layer
- A hidden layer with 100 neurons requires a 784×100 weight matrix
- Each weight in this matrix is a learnable relationship between an input pixel and a hidden feature

This algorithmic structure directly addresses our need for arbitrary feature relationships but creates specific computational patterns that must be handled efficiently by computer systems.

4.2.3 Computational Mapping

The elegant mathematical representation of dense matrix multiplication maps to specific computational patterns that systems must handle. Let's examine how this mapping progresses from mathematical abstraction to computational reality.

The first implementation is shown in Listing 4.1. The function mlp_layer_matrix directly mirrors our mathematical equation. It uses high-level matrix operations (`matmul`) to express the computation in a single line, hiding the

Listing 4.1: Mathematical abstraction in code

```
def mlp_layer_matrix(X, W, b):
    # X: input matrix (batch_size × num_inputs)
    # W: weight matrix (num_inputs × num_outputs)
    # b: bias vector (num_outputs)
    H = activation(matmul(X, W) + b)
    # One clean line of math
    return H
```

underlying complexity. This is the style commonly used in deep learning frameworks, where optimized libraries handle the actual computation.

The second implementation, `mlp_layer_compute` (shown in Listing 4.2), exposes the actual computational pattern through nested loops. This version shows us what really happens when we compute a layer's output: we process each sample in the batch, computing each output neuron by accumulating weighted contributions from all inputs.

Listing 4.2: Core computational pattern

```
def mlp_layer_compute(X, W, b):
    # Process each sample in the batch
    for batch in range(batch_size):
        # Compute each output neuron
        for out in range(num_outputs):
            # Initialize with bias
            Z[batch,out] = b[out]
            # Accumulate weighted inputs
            for in_ in range(num_inputs):
                Z[batch,out] += X[batch,in_] * W[in_,out]

    H = activation(Z)
    return H
```

This translation from mathematical abstraction to concrete computation exposes how dense matrix multiplication decomposes into nested loops of simpler operations. The outer loop processes each sample in the batch, while the middle loop computes values for each output neuron. Within the innermost loop, the system performs repeated multiply-accumulate operations, combining each input with its corresponding weight.

In the MNIST example, each output neuron requires 784 multiply-accumulate operations and at least 1,568 memory accesses (784 for inputs, 784 for weights). While actual implementations use sophisticated optimizations through libraries like BLAS or cuBLAS, these fundamental patterns drive key system design decisions.

4.2.4 System Implications

When analyzing how computational patterns impact computer systems, we typically examine three fundamental dimensions: memory requirements, computation needs, and data movement. This framework enables a systematic analysis of how algorithmic patterns influence system design decisions. We will use this framework for analyzing other network architectures, allowing us to compare and contrast their different characteristics.

4.2.4.1 Memory Requirements

For dense pattern processing, the memory requirements stem from storing and accessing weights, inputs, and intermediate results. In our MNIST example, connecting our 784-dimensional input layer to a hidden layer of 100 neurons requires 78,400 weight parameters. Each forward pass must access all these weights, along with input data and intermediate results. The all-to-all connectivity pattern means there's no inherent locality in these accesses—every output needs every input and its corresponding weights.

These memory access patterns suggest opportunities for optimization through careful data organization and reuse. Modern processors handle these patterns differently; CPUs leverage their cache hierarchy for data reuse, while GPUs employ specialized memory hierarchies designed for high-bandwidth access. Deep learning frameworks abstract these hardware-specific details through optimized matrix multiplication implementations.

4.2.4.2 Computation Needs

The core computation revolves around multiply-accumulate operations[0] arranged in nested loops. Each output value requires as many multiply-accumulates as there are inputs. For MNIST, this means 784 multiply-accumulates per output neuron. With 100 neurons in our hidden layer, we're performing 78,400 multiply-accumulates for a single input image. While these operations are simple, their volume and arrangement create specific demands on processing resources.

This computational structure lends itself to particular optimization strategies in modern hardware. The dense matrix multiplication pattern can be efficiently parallelized across multiple processing units, with each handling different subsets of neurons. Modern hardware accelerators take advantage of this through specialized matrix multiplication units, while deep learning frameworks automatically convert these operations into optimized BLAS (Basic Linear Algebra Subprograms) calls. CPUs and GPUs can both exploit cache locality by carefully tiling the computation to maximize data reuse, though their specific approaches differ based on their architectural strengths.

4.2.4.3 Data Movement

The all-to-all connectivity pattern in MLPs creates significant data movement requirements. Each multiply-accumulate operation needs three pieces of data: an input value, a weight value, and the running sum. For our MNIST example layer, computing a single output value requires moving 784 inputs and 784

[0] Multiply-Accumulate Operation: A basic operation in digital computing and neural networks that multiplies two numbers and adds the result to an accumulator.

weights to wherever the computation occurs. This movement pattern repeats for each of the 100 output neurons, creating substantial data transfer demands between memory and compute units.

The predictable nature of these data movement patterns enables strategic data staging and transfer optimizations. Different architectures address this challenge through various mechanisms; CPUs use sophisticated prefetching and multi-level caches; meanwhile, GPUs employ high-bandwidth memory systems and latency hiding through massive threading. Deep learning frameworks orchestrate these data movements through optimized memory management systems.

4.3 Convolutional Neural Networks: Spatial Pattern Processing

While MLPs treat each input element independently, many real-world data types exhibit strong spatial relationships. Images, for example, derive their meaning from the spatial arrangement of pixels—a pattern of edges and textures that form recognizable objects. Audio signals show temporal patterns of frequency components, and sensor data often contains spatial or temporal correlations. These spatial relationships suggest that treating every input-output connection with equal importance, as MLPs do, might not be the most effective approach.

4.3.1 Pattern Processing Needs

Spatial pattern processing addresses scenarios where the relationship between data points depends on their relative positions or proximity. Consider processing a natural image: a pixel's relationship with its neighbors is important for detecting edges, textures, and shapes. These local patterns then combine hierarchically to form more complex features—edges form shapes, shapes form objects, and objects form scenes.

This hierarchical spatial pattern processing appears across many domains. In computer vision, local pixel patterns form edges and textures that combine into recognizable objects. Speech processing relies on patterns across nearby time segments to identify phonemes and words. Sensor networks analyze correlations between physically proximate sensors to understand environmental patterns. Medical imaging depends on recognizing tissue patterns that indicate biological structures.

Taking image processing as an example, if we want to detect a cat in an image, certain spatial patterns must be recognized: the triangular shape of ears, the round contours of the face, the texture of fur. Importantly, these patterns maintain their meaning regardless of where they appear in the image—a cat is still a cat whether it's in the top-left or bottom-right corner. This suggests two key requirements for spatial pattern processing: the ability to detect local patterns and the ability to recognize these patterns regardless of their position.

This leads us to the convolutional neural network architecture (CNN), introduced by Y. LeCun et al. (1989). As illustrated in Figure 4.3, CNNs address

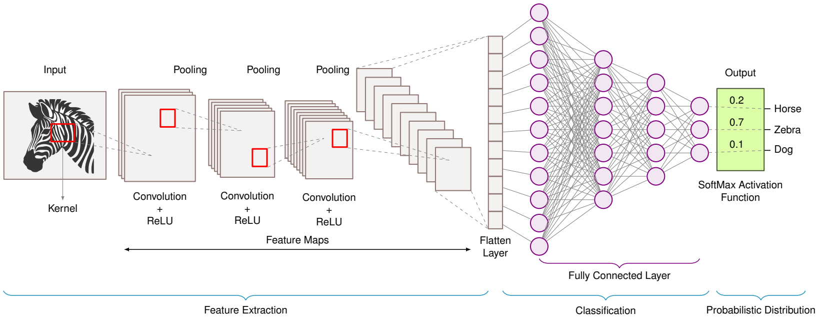


Figure 4.3: How convolutional neural networks extract spaital features for classification.

spatial pattern processing through a fundamentally different connection pattern than MLPs. Instead of connecting every input to every output, CNNs use a local connection pattern where each output connects only to a small, spatially contiguous region of the input. This local receptive field moves across the input space, applying the same set of weights at each position—a process known as convolution[1].

4.3.2 Algorithmic Structure

The core operation in a CNN can be expressed mathematically as:

$$\mathbf{H}^{(l)}_{i,j,k} = f\left(\sum_{di}\sum_{dj}\sum_{c}\mathbf{W}^{(l)}_{di,dj,c,k}\mathbf{H}^{(l-1)}_{i+di,j+dj,c} + \mathbf{b}^{(l)}_{k}\right)$$

Here, (i,j) corresponds to spatial positions, k indexes output channels, c indexes input channels, and (di,dj) spans the local receptive field. Unlike the dense matrix multiplication of MLPs, this operation:

- Processes local neighborhoods (typically 3×3 or 5×5)
- Reuses the same weights at each spatial position
- Maintains spatial structure in its output

For a concrete example, consider the MNIST digit classification task with 28×28 grayscale images. Each convolutional layer applies a set of filters (e.g., 3×3) that slide across the image, computing local weighted sums. If we use 32 filters, the layer produces a $28\times 28\times 32$ output, where each spatial position contains 32 different feature measurements of its local neighborhood. This contrasts sharply with the multi-layer perceptron (MLP) approach, where the entire image is flattened into a 784-dimensional vector before processing.

This algorithmic structure directly implements the requirements for spatial pattern processing, creating distinct computational patterns that influence system design. Unlike MLPs, convolutional networks preserve spatial locality, allowing for efficient hierarchical feature extraction. These properties drive architectural optimizations in AI accelerators, where operations such as data reuse, tiling, and parallel filter computation are critical for performance.

As illustrated in Figure 4.4, convolution operations involve sliding a small filter over the input image to generate a feature map. This process efficiently

[1] Convolution: A mathematical operation on two functions producing a third function expressing how the shape of one is modified by the other.

captures local structures while maintaining translation invariance, making it a fundamental component of modern deep learning architectures. For an interactive visual exploration of convolutional networks, the CNN Explainer project provides an insightful demonstration of how these networks are constructed.

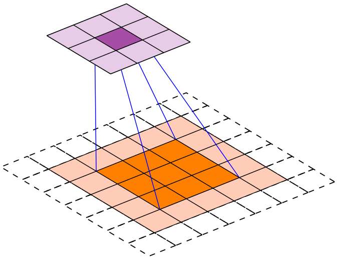


Figure 4.4: Convolution operation, image data (blue) and 3×3 filter (green). Source: V. Dumoulin, F. Visin, MIT

4.3.3 Computational Mapping

The elegant spatial structure of convolution operations maps to computational patterns quite different from the dense matrix multiplication of MLPs. Let's examine how this mapping progresses from mathematical abstraction to computational reality.

The first implementation, `conv_layer_spatial` (shown in Listing 4.3), uses high-level convolution operations to express the computation concisely. This is typical in deep learning frameworks, where optimized libraries handle the underlying complexity.

Listing 4.3: Mathematical abstraction - simple and clean

```
def conv_layer_spatial(input, kernel, bias):
    output = convolution(input, kernel) + bias
    return activation(output)
```

The second implementation, conv_layer_compute (see Listing 4.4), reveals the actual computational pattern: nested loops that process each spatial position, applying the same filter weights to local regions of the input. These nested loops reveal the true nature of convolution's computational structure.

The seven nested loops reveal different aspects of the computation:

- Outer loops (1-3) manage position: which image and where in the image
- Middle loop (4) handles output features: computing different learned patterns
- Inner loops (5-7) perform the actual convolution: sliding the kernel window

Let's take a closer look. The outer two loops (`for y` and `for x`) traverse each spatial position in the output feature map (for our MNIST example, this means moving across all 28×28 positions). At each position, we compute values for

Listing 4.4: System reality - nested loops of computation

```
def conv_layer_compute(input, kernel, bias):
  # Loop 1: Process each image in batch
  for image in range(batch_size):

    # Loop 2&3: Move across image spatially
    for y in range(height):
      for x in range(width):

       # Loop 4: Compute each output feature
       for out_channel in range(num_output_channels):
         result = bias[out_channel]

         # Loop 5&6: Move across kernel window
         for ky in range(kernel_height):
           for kx in range(kernel_width):

             # Loop 7: Process each input feature
             for in_channel in range(num_input_channels):
             # Get input value from correct window position
              in_y = y + ky
              in_x = x + kx
              # Perform multiply-accumulate operation
              result += (
                input[image, in_y, in_x, in_channel]
                * kernel[ky, kx, in_channel, out_channel]
              )

         # Store result for this output position
         output[image, y, x, out_channel] = result
```

each output channel (`for k` loop), which represents different learned features or patterns—our 32 different feature detectors.

The inner three loops implement the actual convolution operation at each position. For each output value, we process a local 3×3 region of the input (the `dy` and `dx` loops) across all input channels (`for c` loop). This creates a sliding window effect, where the same 3×3 filter moves across the image, performing multiply-accumulates between the filter weights and the local input values. Unlike the MLP's global connectivity, this local processing pattern means each output value depends only on a small neighborhood of the input.

For our MNIST example with 3×3 filters and 32 output channels, each output position requires only 9 multiply-accumulate operations per input channel, compared to the 784 operations needed in our MLP layer. However, this operation must be repeated for every spatial position (28×28) and every output channel (32).

While using fewer operations per output, the spatial structure creates different patterns of memory access and computation that systems must handle efficiently. These patterns fundamentally influence system design, creating both challenges and opportunities for optimization, which we'll examine next.

4.3.4 System Implications

When analyzing how computational patterns impact computer systems, we examine three fundamental dimensions: memory requirements, computation needs, and data movement. For CNNs, the spatial nature of processing creates distinctive patterns in each dimension that differ significantly from the dense connectivity of MLPs.

4.3.4.1 Memory Requirements

For convolutional layers, memory requirements center around two key components: filter weights and feature maps[2]. Unlike MLPs that require storing full connection matrices, CNNs use small, reusable filters. In our MNIST example, a convolutional layer with 32 filters of size 3×3 requires storing only 288 weight parameters $(3 \times 3 \times 32)$, in contrast to the 78,400 weights needed for our MLP's fully-connected layer. However, the system must store feature maps for all spatial positions, creating a different memory demand—a 28×28 input with 32 output channels requires storing 25,088 activation values $(28 \times 28 \times 32)$.

These memory access patterns suggest opportunities for optimization through weight reuse and careful feature map management. Modern processors handle these patterns by caching filter weights, which are reused across spatial positions, while streaming through feature map data. Deep learning frameworks typically implement this through specialized memory layouts that optimize for both filter reuse and spatial locality in feature map access. CPUs and GPUs approach this differently—CPUs leverage their cache hierarchy to keep frequently used filters resident, while GPUs use specialized memory architectures designed for the spatial access patterns of image processing.

4.3.4.2 Computation Needs

The core computation in CNNs involves repeatedly applying small filters across spatial positions. Each output value requires a local multiply-accumulate operation over the filter region. For our MNIST example with 3×3 filters and 32 output channels, computing one spatial position involves 288 multiply-accumulates $(3 \times 3 \times 32)$, and this must be repeated for all 784 spatial positions (28×28). While each individual computation involves fewer operations than an MLP layer, the total computational load remains substantial due to spatial repetition.

This computational pattern presents different optimization opportunities than MLPs. The regular, repeated nature of convolution operations enables efficient hardware utilization through structured parallelism. Modern processors exploit this pattern in various ways. CPUs leverage SIMD[3] instructions to process multiple filter positions simultaneously, while GPUs parallelize computation across spatial positions and channels. Deep learning frameworks further

[2] Feature Map: The output of one layer of a neural network, which serves as the input for the next layer.

[3] Single Instruction, Multiple Data (SIMD): A type of parallel computing used in processors.

optimize this through specialized convolution algorithms that transform the computation to better match hardware capabilities.

4.3.4.3 Data Movement

The sliding window pattern of convolutions creates a distinctive data movement profile. Unlike MLPs where each weight is used once per forward pass, CNN filter weights are reused many times as the filter slides across spatial positions. For our MNIST example, each 3×3 filter weight is reused 784 times (once for each position in the 28×28 feature map). However, this creates a different challenge: the system must stream input features through the computation unit while keeping filter weights stable.

The predictable spatial access pattern enables strategic data movement optimizations. Different architectures handle this movement pattern through specialized mechanisms. CPUs maintain frequently used filter weights in cache while streaming through input features. GPUs employ memory architectures optimized for spatial locality and provide hardware support for efficient sliding window operations. Deep learning frameworks orchestrate these movements by organizing computations to maximize filter weight reuse and minimize redundant feature map accesses.

4.4 Recurrent Neural Networks: Sequential Pattern Processing

While MLPs handle arbitrary relationships and CNNs process spatial patterns, many real-world problems involve sequential data where the order and relationship between elements over time matters. Text processing requires understanding how words relate to previous context, speech recognition needs to track how sounds form coherent patterns, and time-series analysis must capture how values evolve over time. These sequential relationships suggest that treating each time step independently misses crucial temporal patterns.

4.4.1 Pattern Processing Needs

Sequential pattern processing addresses scenarios where the meaning of current input depends on what came before it. Consider natural language processing: the meaning of a word often depends heavily on previous words in the sentence. The word “bank” means something different in “river bank” versus “bank account.” Similarly, in speech recognition, a phoneme’s interpretation often depends on surrounding sounds, and in financial forecasting, future predictions require understanding patterns in historical data.

The key challenge in sequential processing is maintaining and updating relevant context over time. When reading text, humans don’t start fresh with each word—we maintain a running understanding that evolves as we process new information. Similarly, when processing time-series data, patterns might span different timescales, from immediate dependencies to long-term trends. This suggests we need an architecture that can both maintain state over time and update it based on new inputs.

These requirements demand specific capabilities from our processing architecture. The system must maintain internal state to capture temporal context, update this state based on new inputs, and learn which historical information is relevant for current predictions. Unlike MLPs and CNNs, which process fixed-size inputs, sequential processing must handle variable-length sequences while maintaining computational efficiency. This leads us to the recurrent neural network (RNN) architecture.

4.4.2 Algorithmic Structure

RNNs address sequential processing through a fundamentally different approach than MLPs or CNNs by introducing recurrent connections. Instead of just mapping inputs to outputs, RNNs maintain an internal state that is updated at each time step. This creates a memory mechanism that allows the network to carry information forward in time. This unique ability to model temporal dependencies was first explored by Elman (2002), who demonstrated how RNNs could find structure in time-dependent data.

The core operation in a basic RNN can be expressed mathematically as:

$$\mathbf{h}_t = f(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)$$

where $\mathbf{h}_t$ corresponds to the hidden state at time t, $\mathbf{x}_t$ is the input at time t, $\mathbf{W}_{hh}$ contains the recurrent weights, and $\mathbf{W}_{xh}$ contains the input weights, as shown in the unfolded network structure in Figure 4.5.

For example, in processing a sequence of words, each word might be represented as a 100-dimensional vector ($\mathbf{x}_t$), and we might maintain a hidden state of 128 dimensions ($\mathbf{h}_t$). At each time step, the network combines the current input with its previous state to update its understanding of the sequence. This creates a form of memory that can capture patterns across time steps.

This recurrent structure directly implements our requirements for sequential processing through the introduction of recurrent connections, which maintain internal state and allow the network to carry information forward in time. Instead of processing all inputs independently, RNNs process sequences of data by iteratively updating a hidden state based on the current input and the previous hidden state, as depicted in Figure 4.5. This makes RNNs well-suited for tasks such as language modeling, speech recognition, and time-series forecasting.

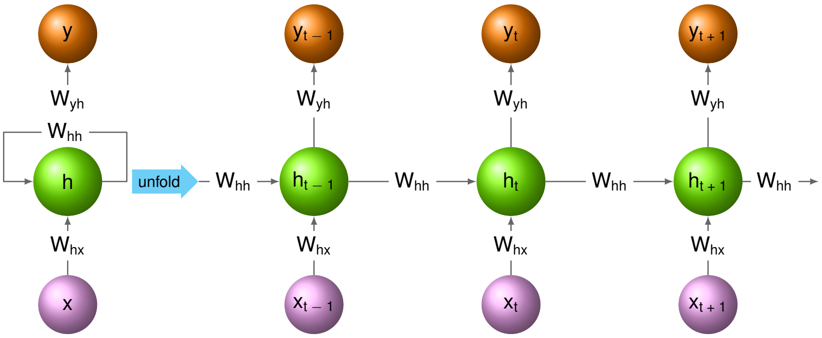


Figure 4.5: RNN architecture. Source: A. Amidi, S. Amidi, Stanford

4.4.3 Computational Mapping

The sequential structure of RNNs maps to computational patterns quite different from both MLPs and CNNs. Let's examine how this mapping progresses from mathematical abstraction to computational reality.

As shown in Listing 4.5, the `rnn_layer_step` function demonstrates how the operation looks using high-level matrix operations found in deep learning frameworks. It handles a single time step, taking the current input `x_t` and previous hidden state `h_prev`, along with two weight matrices: `W_hh` for hidden-to-hidden connections and `W_xh` for input-to-hidden connections. Through matrix multiplication operations (`matmul`), it merges the previous state and current input to generate the next hidden state.

Listing 4.5: Mathematical abstraction in code

```
def rnn_layer_step(x_t, h_prev, W_hh, W_xh, b):
  # x_t: input at time t (batch_size × input_dim)
  # h_prev: previous hidden state (batch_size × hidden_dim)
  # W_hh: recurrent weights (hidden_dim × hidden_dim)
  # W_xh: input weights (input_dim × hidden_dim)
  h_t = activation(
    matmul(h_prev, W_hh)
    + matmul(x_t, W_xh)
    + b
  )
  return h_t
```

This simplified view masks the underlying complexity of the nested loops and individual computations shown in the detailed implementation (Listing 4.6). Its actual implementation reveals a more detailed computational reality.

The nested loops in `rnn_layer_compute` expose the core computational pattern of RNNs (see Listing 4.6). Loop 1 processes each sequence in the batch independently, allowing for batch-level parallelism. Within each batch item, Loop 2 computes how the previous hidden state influences the next state through the recurrent weights `W_hh`. Loop 3 then incorporates new information from the current input through the input weights `W_xh`. Finally, Loop 4 adds biases and applies the activation function to produce the new hidden state.

For a sequence processing task with input dimension 100 and hidden state dimension 128, each time step requires two matrix multiplications: one 128×128 for the recurrent connection and one 100×128 for the input projection. While individual time steps can process in parallel across batch elements, the time steps themselves must process sequentially. This creates a unique computational pattern that systems must handle efficiently.

Listing 4.6: Core computational pattern

```
def rnn_layer_compute(x_t, h_prev, W_hh, W_xh, b):
    # Initialize next hidden state
    h_t = np.zeros_like(h_prev)

    # Loop 1: Process each sequence in the batch
    for batch in range(batch_size):
        # Loop 2: Compute recurrent contribution
        # (h_prev × W_hh)
        for i in range(hidden_dim):
            for j in range(hidden_dim):
                h_t[batch,i] += h_prev[batch,j] * W_hh[j,i]

        # Loop 3: Compute input contribution (x_t × W_xh)
        for i in range(hidden_dim):
            for j in range(input_dim):
                h_t[batch,i] += x_t[batch,j] * W_xh[j,i]

        # Loop 4: Add bias and apply activation
        for i in range(hidden_dim):
            h_t[batch,i] = activation(h_t[batch,i] + b[i])

    return h_t
```

4.4.4 System Implications

For RNNs, the sequential nature of processing creates distinctive patterns in each dimension (memory requirements, computation needs, and data movement) that differ significantly from both MLPs and CNNs.

4.4.4.1 Memory Requirements

RNNs require storing two sets of weights (input-to-hidden and hidden-to-hidden) along with the hidden state. For our example with input dimension 100 and hidden state dimension 128, this means storing 12,800 weights for input projection (100×128) and 16,384 weights for recurrent connections (128×128). Unlike CNNs where weights are reused across spatial positions, RNN weights are reused across time steps. Additionally, the system must maintain the hidden state, which becomes a critical factor in memory usage and access patterns.

These memory access patterns create a different profile from MLPs and CNNs. Modern processors handle these patterns by keeping the weight matrices in cache[4] while streaming through sequence elements. Deep learning frameworks optimize memory access by batching sequences together and carefully managing hidden state storage between time steps. CPUs and GPUs approach this through different strategies; CPUs leverage their cache hierarchy for weight reuse; meanwhile, GPUs use specialized memory architectures designed for maintaining state across sequential operations.

[4] Memory storage area where frequently accessed data can be stored for rapid access.

4.4.4.2 Computation Needs

The core computation in RNNs involves repeatedly applying weight matrices across time steps. For each time step, we perform two matrix multiplications: one with the input weights and one with the recurrent weights. In our example, processing a single time step requires 12,800 multiply-accumulates for the input projection (100×128) and 16,384 multiply-accumulates for the recurrent connection (128×128).

This computational pattern differs from both MLPs and CNNs in a key way: while we can parallelize across batch elements, we cannot parallelize across time steps due to the sequential dependency. Each time step must wait for the previous step's hidden state before it can begin computation. This creates a tension between the inherent sequential nature of the algorithm and the desire for parallel execution in modern hardware.

Modern processors handle these patterns through different approaches. CPUs pipeline operations within each time step while maintaining the sequential order across steps. GPUs batch multiple sequences together to maintain high throughput despite sequential dependencies. Deep learning frameworks optimize this further by techniques like sequence packing[5] and unrolling computations across multiple time steps when possible.

4.4.4.3 Data Movement

The sequential processing in RNNs creates a distinctive data movement pattern that differs from both MLPs and CNNs. While MLPs need each weight only once per forward pass and CNNs reuse weights across spatial positions, RNNs reuse their weights across time steps while requiring careful management of the hidden state data flow.

For our example with a 128-dimensional hidden state, each time step must: load the previous hidden state (128 values), access both weight matrices (29,184 total weights from both input and recurrent connections), and store the new hidden state (128 values). This pattern repeats for every element in the sequence. Unlike CNNs where we can predict and prefetch data based on spatial patterns, RNN data movement is driven by temporal dependencies.

Different architectures handle this sequential data movement through specialized mechanisms. CPUs maintain weight matrices in cache while streaming through sequence elements and managing hidden state updates. GPUs employ memory architectures optimized for maintaining state information across sequential operations while processing multiple sequences in parallel. Deep learning frameworks orchestrate these movements by managing data transfers between time steps and optimizing batch operations.

4.5 Attention Mechanisms: Dynamic Pattern Processing

While previous architectures process patterns in fixed ways, such as MLPs with dense connectivity, CNNs with spatial operations, and RNNs with sequential updates, many tasks require dynamic relationships between elements that change based on content. Language understanding, for instance, needs to capture relationships between words that depend on meaning rather than just

[5] Sequence Packing: A technique in deep learning where sequences of different lengths are packed together to optimize memory and processing efficiency.

position. Graph analysis requires understanding connections that vary by node. These dynamic relationships suggest we need an architecture that can learn and adapt its processing patterns based on the data itself.

4.5.1 Pattern Processing Needs

Dynamic pattern processing addresses scenarios where relationships between elements aren't fixed by architecture but instead emerge from content. Consider language translation: when translating "the bank by the river," understanding "bank" requires attending to "river," but in "the bank approved the loan," the important relationship is with "approved" and "loan." Unlike RNNs that process information sequentially or CNNs that use fixed spatial patterns, we need an architecture that can dynamically determine which relationships matter.

This requirement for dynamic processing appears across many domains. In protein structure prediction, interactions between amino acids depend on their chemical properties and spatial arrangements. In graph analysis, node relationships vary based on graph structure and node features. In document analysis, connections between different sections depend on semantic content rather than just proximity.

These scenarios demand specific capabilities from our processing architecture. The system must compute relationships between all pairs of elements, weigh these relationships based on content, and use these weights to selectively combine information. Unlike previous architectures with fixed connectivity patterns, dynamic processing requires the flexibility to modify its computation graph based on the input itself. This leads us to the Transformer architecture, which implements these capabilities through attention mechanisms. Figure 4.6 shows the relationships learned for an attention head between subwords in a sentence.

4.5.2 Basic Attention Mechanism

4.5.2.1 Algorithmic Structure

Attention mechanisms form the foundation of dynamic pattern processing by computing weighted connections between elements based on their content (Bahdanau, Cho, and Bengio 2014). This approach allows for the processing of relationships that aren't fixed by architecture but instead emerge from the data itself. At the core of an attention mechanism is a fundamental operation that can be expressed mathematically as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

In this equation, **Q** (queries), **K** (keys), and **V** (values) represent learned projections of the input. For a sequence of length N with dimension d, this operation creates an $N \times N$ attention matrix, determining how each position should attend to all others.

The attention operation involves several key steps. First, it computes query, key, and value projections for each position in the sequence. Next, it generates

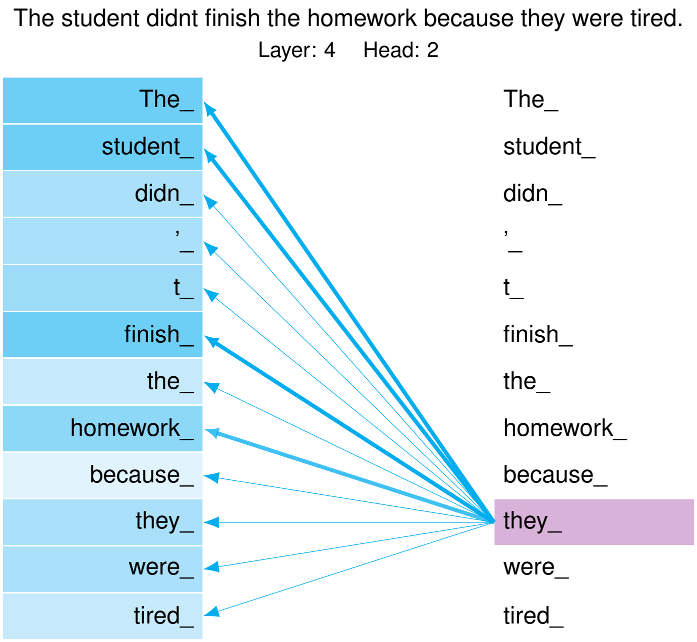


Figure 4.6: Transformer architectures "attend" or identify pairwise relationships with subwords in a sequence.

an $N \times N$ attention matrix through query-key interactions. These steps are illustrated in Figure 4.7. Finally, it uses these attention weights to combine value vectors, producing the output.

The key is that, unlike the fixed weight matrices found in previous architectures, as shown in Figure 4.8, these attention weights are computed dynamically for each input. This allows the model to adapt its processing based on the dynamic content at hand.

4.5.2.2 Computational Mapping

The dynamic structure of attention operations maps to computational patterns that differ significantly from those of previous architectures. To understand this mapping, let's examine how it progresses from mathematical abstraction to computational reality (see Listing 4.7).

The nested loops in `attention_layer_compute` reveal the true nature of attention's computational pattern (see Listing 4.7). The first loop processes each sequence in the batch independently. The second and third loops compute attention scores between all pairs of positions, creating a quadratic computation pattern with respect to sequence length. The fourth loop uses these attention weights to combine values from all positions, producing the final output.

4.5.2.3 System Implications

The attention mechanism creates distinctive patterns in memory requirements, computation needs, and data movement that set it apart from previous architectures.

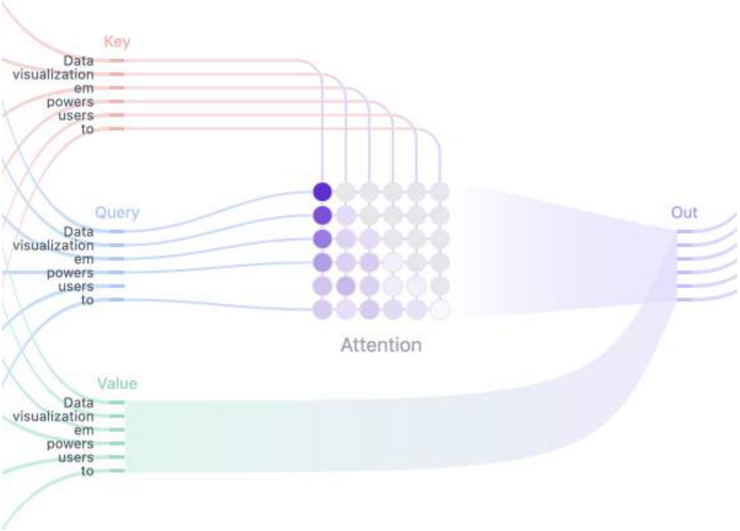


Figure 4.7: The interaction between Query, Key, and Value components. Source: Transformer Explainer.

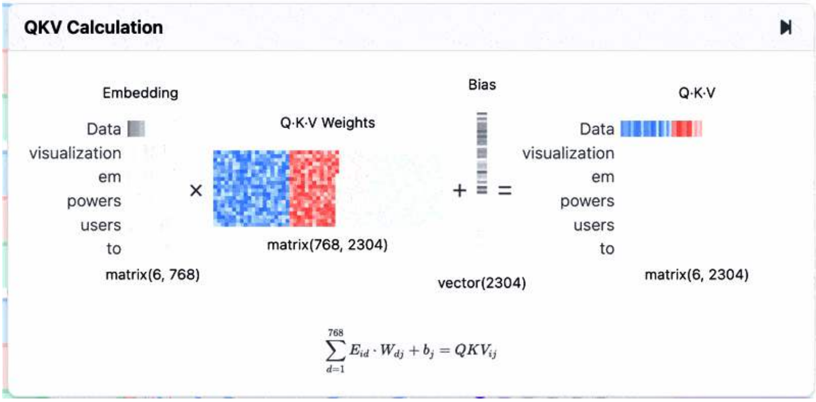


Figure 4.8: Dynamic weight calculation. Source: Transformer Explainer.

Listing 4.7: Mathematical abstraction in code

```python
def attention_layer_matrix(Q, K, V):
    # Q, K, V: (batch_size × seq_len × d_model)
    scores = matmul(Q, K.transpose(-2, -1)) / \
             sqrt(d_k)           # Compute attention scores
    weights = softmax(scores)    # Normalize scores
    output = matmul(weights, V)  # Combine values
    return output

# Core computational pattern
def attention_layer_compute(Q, K, V):
    # Initialize outputs
    scores = np.zeros((batch_size, seq_len, seq_len))
    outputs = np.zeros_like(V)

    # Loop 1: Process each sequence in batch
    for b in range(batch_size):
        # Loop 2: Compute attention for each query position
        for i in range(seq_len):
            # Loop 3: Compare with each key position
            for j in range(seq_len):
                # Compute attention score
                for d in range(d_model):
                    scores[b,i,j] += Q[b,i,d] * K[b,j,d]
                scores[b,i,j] /= sqrt(d_k)

        # Apply softmax to scores
        for i in range(seq_len):
            scores[b,i] = softmax(scores[b,i])

        # Loop 4: Combine values using attention weights
        for i in range(seq_len):
            for j in range(seq_len):
                for d in range(d_model):
                    outputs[b, i, d] += (
                       scores[b, i, j]
                       * V[b, j, d]
                    )

    return outputs
```

Memory Requirements. In terms of memory requirements, attention mechanisms necessitate storage for attention weights, key-query-value projections, and intermediate feature representations. For a sequence length N and dimension d, each attention layer must store an $N \times N$ attention weight matrix for each sequence in the batch, three sets of projection matrices for queries, keys, and values (each sized $d \times d$), and input and output feature maps of size $N \times d$. The dynamic generation of attention weights for every input creates a memory access pattern where intermediate attention weights become a significant factor in memory usage.

Computation Needs. Computation needs in attention mechanisms center around two main phases: generating attention weights and applying them to values. For each attention layer, the system performs substantial multiply-accumulate operations across multiple computational stages. The query-key interactions alone require $N \times N \times d$ multiply-accumulates, with an equal number needed for applying attention weights to values. Additional computations are required for the projection matrices and softmax operations. This computational pattern differs from previous architectures due to its quadratic scaling with sequence length and the need to perform fresh computations for each input.

Data Movement. Data movement in attention mechanisms presents unique challenges. Each attention operation involves projecting and moving query, key, and value vectors for each position, storing and accessing the full attention weight matrix, and coordinating the movement of value vectors during the weighted combination phase. This creates a data movement pattern where intermediate attention weights become a major factor in system bandwidth requirements. Unlike the more predictable access patterns of CNNs or the sequential access of RNNs, attention operations require frequent movement of dynamically computed weights across the memory hierarchy.

These distinctive characteristics of attention mechanisms in terms of memory, computation, and data movement have significant implications for system design and optimization, setting the stage for the development of more advanced architectures like Transformers.

4.5.3 Transformers and Self-Attention

Transformers, first introduced by M. X. Chen et al. (2018), represent a significant evolution in the application of attention mechanisms, introducing the concept of self-attention to create a powerful architecture for dynamic pattern processing. While the basic attention mechanism allows for content-based weighting of information from a source sequence, Transformers extend this idea by applying attention within a single sequence, enabling each element to attend to all other elements including itself.

4.5.3.1 Algorithmic Structure

The key innovation in Transformers lies in their use of self-attention layers. In a self-attention layer, the queries, keys, and values are all derived from the same input sequence. This allows the model to weigh the importance of different positions within the same sequence when encoding each position. For instance, in processing the sentence “The animal didn’t cross the street because it was too wide,” self-attention allows the model to link “it” with “street,” capturing long-range dependencies that are challenging for traditional sequential models.

Transformers typically employ multi-head attention, which involves multiple sets of query/key/value projections. Each set, or "head," can focus on different aspects of the input, allowing the model to jointly attend to information from different representation subspaces. This multi-head structure provides the model with a richer representational capability, enabling it to capture various types of relationships within the data simultaneously.

The self-attention mechanism in Transformers can be expressed mathematically in a form similar to the basic attention mechanism:

$$\text{SelfAttention}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{XW_Q}(\mathbf{XW_K})^T}{\sqrt{d_k}}\right)\mathbf{XW_V}$$

Here, $\mathbf{X}$ is the input sequence, and $\mathbf{W_Q}$, $\mathbf{W_K}$, and $\mathbf{W_V}$ are learned weight matrices for queries, keys, and values respectively. This formulation highlights how self-attention derives all its components from the same input, creating a dynamic, content-dependent processing pattern.

The Transformer architecture leverages this self-attention mechanism within a broader structure that typically includes feed-forward layers, layer normalization, and residual connections (see Figure 4.9). This combination allows Transformers to process input sequences in parallel, capturing complex dependencies without the need for sequential computation. As a result, Transformers have demonstrated remarkable effectiveness across a wide range of tasks, from natural language processing to computer vision, revolutionizing the landscape of deep learning architectures.

4.5.3.2 Computational Mapping

While Transformer self-attention builds upon the basic attention mechanism, it introduces distinct computational patterns that set it apart. To understand these patterns, we must examine the typical implementation of self-attention in Transformers (see Listing 4.8):

4.5.3.3 System Implications

This implementation reveals several key computational characteristics of Transformer self-attention. First, self-attention enables parallel processing across all positions in the sequence. This is evident in the matrix multiplications that compute `Q`, `K`, and `V` simultaneously for all positions. Unlike recurrent architectures that process inputs sequentially, this parallel nature allows for more efficient computation, especially on modern hardware designed for parallel operations.

Second, the attention score computation results in a matrix of size `(seq_len × seq_len)`, leading to quadratic complexity with respect to sequence length. This quadratic relationship becomes a significant computational bottleneck when processing long sequences, a challenge that has spurred research into more efficient attention mechanisms.

Third, the multi-head attention mechanism effectively runs multiple self-attention operations in parallel, each with its own set of learned projections. While this increases the computational load linearly with the number of heads,

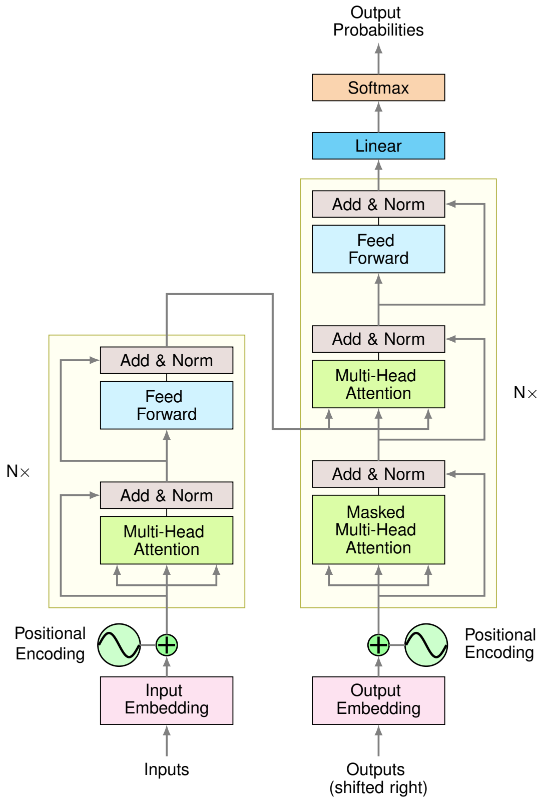


Figure 4.9: The Transformer model architecture. Source: Attention Is All You Need

it allows the model to capture different types of relationships within the same input, enhancing the model's representational power.

Fourth, the core computations in self-attention are dominated by large matrix multiplications. For a sequence of length N and embedding dimension d, the main operations involve matrices of sizes $(N \times d)$, $(d \times d)$, and $(N \times N)$. These intensive matrix operations are well-suited for acceleration on specialized hardware like GPUs, but they also contribute significantly to the overall computational cost of the model.

Finally, self-attention generates memory-intensive intermediate results. The attention weights matrix $(N \times N)$ and the intermediate results for each attention head create substantial memory requirements, especially for long sequences. This can pose challenges for deployment on memory-constrained devices and necessitates careful memory management in implementations.

These computational patterns create a unique profile for Transformer self-attention, distinct from previous architectures. The parallel nature of the computations makes Transformers well-suited for modern parallel processing hardware, but the quadratic complexity with sequence length poses challenges for processing long sequences. As a result, much research has focused on

Listing 4.8: Self-attention mechanism in Transformers

```
def self_attention_layer(X, W_Q, W_K, W_V, d_k):
    # X: input tensor (batch_size × seq_len × d_model)
    # W_Q, W_K, W_V: weight matrices (d_model × d_k)

    Q = matmul(X, W_Q)
    K = matmul(X, W_K)
    V = matmul(X, W_V)

    scores = matmul(Q, K.transpose(-2, -1)) / sqrt(d_k)
    attention_weights = softmax(scores, dim=-1)
    output = matmul(attention_weights, V)

    return output

def multi_head_attention(
    X, W_Q, W_K, W_V, W_O, num_heads, d_k
):
    outputs = []
    for i in range(num_heads):
        head_output = self_attention_layer(
            X, W_Q[i], W_K[i], W_V[i], d_k
        )
        outputs.append(head_output)

    concat_output = torch.cat(outputs, dim=-1)
    final_output = matmul(concat_output, W_O)

    return final_output
```

developing optimization techniques, such as sparse attention patterns or low-rank approximations, to address these challenges. Each of these optimizations presents its own trade-offs between computational efficiency and model expressiveness, a balance that must be carefully considered in practical applications.

4.6 Architectural Building Blocks

Deep learning architectures, while we presented them as distinct approaches in the previous sections, are better understood as compositions of fundamental building blocks that evolved over time. Much like how complex LEGO structures are built from basic bricks, modern neural networks combine and iterate on core computational patterns that emerged through decades of research (Yann LeCun, Bengio, and Hinton 2015a). Each architectural innovation introduced new building blocks while finding novel ways to use existing ones.

These building blocks and their evolution provide insight into modern architectures. What began with the simple perceptron (Rosenblatt 1958) evolved into multi-layer networks (Rumelhart, Hinton, and Williams 1986), which then spawned specialized patterns for spatial and sequential processing. Each advancement maintained useful elements from its predecessors while introducing new computational primitives. Today's sophisticated architectures, like Transformers, can be seen as carefully engineered combinations of these fundamental building blocks.

This progression reveals not just the evolution of neural networks, but also the discovery and refinement of core computational patterns that remain relevant. As we have seen through our exploration of different neural network architectures, deep learning has evolved significantly, with each new architecture bringing its own set of computational demands and system-level challenges.

Table 4.1 summarizes this evolution, highlighting the key primitives and system focus for each era of deep learning development. This table encapsulates the major shifts in deep learning architecture design and the corresponding changes in system-level considerations. From the early focus on dense matrix operations optimized for CPUs, we see a progression through convolutions leveraging GPU acceleration, to sequential operations necessitating sophisticated memory hierarchies, and finally to the current era of attention mechanisms requiring flexible accelerators and high-bandwidth memory.

Table 4.1: Evolution of deep learning architectures and their system implications

Era	Dominant Architecture	Key Primitives	System Focus
Early NN	MLP	Dense Matrix Ops	CPU optimization
CNN Revolution	CNN	Convolutions	GPU acceleration
Sequence Modeling	RNN	Sequential Ops	Memory hierarchies
Attention Era	Transformer	Attention, Dynamic Compute	Flexible accelerators, High-bandwidth memory

As we dive deeper into each of these building blocks, we see how these primitives evolved and combined to create increasingly powerful and complex neural network architectures.

4.6.1 From Perceptron to Multi-Layer Networks

While we examined MLPs earlier as a mechanism for dense pattern processing, here we focus on how they established fundamental building blocks that appear throughout deep learning. The evolution from perceptron to MLP introduced several key concepts: the power of layer stacking, the importance of non-linear transformations, and the basic feedforward computation pattern.

The introduction of hidden layers between input and output created a template for feature transformation that appears in virtually every modern architecture. Even in sophisticated networks like Transformers, we find MLP-style feedforward layers performing feature processing. The concept of transforming data through successive non-linear layers has become a fundamental paradigm that transcends the specific architecture types.

Perhaps most importantly, the development of MLPs established the backpropagation algorithm, which to this day remains the cornerstone of neural network training. This key contribution has enabled the training of deep architectures and influenced how later architectures would be designed to maintain gradient flow.

These building blocks, layered feature transformation, non-linear activation, and gradient-based learning, set the foundation for more specialized architectures. Subsequent innovations often focused on structuring these basic components in new ways rather than replacing them entirely.

4.6.2 From Dense to Spatial Processing

The development of CNNs marked a significant architectural innovation, specifically the realization that we could specialize the dense connectivity of MLPs for spatial patterns. While retaining the core concept of layer-wise processing, CNNs introduced several fundamental building blocks that would influence all future architectures.

The first key innovation was the concept of parameter sharing. Unlike MLPs where each connection had its own weight, CNNs showed how the same parameters could be reused across different parts of the input. This not only made the networks more efficient but introduced the powerful idea that architectural structure could encode useful priors about the data (Lecun et al. 1998).

Perhaps even more influential was the introduction of skip connections through ResNets (K. He et al. 2016a). Originally they were designed to help train very deep CNNs, skip connections have become a fundamental building block that appears in virtually every modern architecture. They showed how direct paths through the network could help gradient flow and information propagation, a concept now central to Transformer designs.

CNNs also introduced batch normalization, a technique for stabilizing neural network training by normalizing intermediate features (Ioffe and Szegedy 2015a); we will learn more about this in the AI Training chapter. This concept of feature normalization, while originating in CNNs, evolved into layer normalization and is now a key component in modern architectures.

These innovations, such as parameter sharing, skip connections, and normalization, transcended their origins in spatial processing to become essential building blocks in the deep learning toolkit.

4.6.3 The Evolution of Sequence Processing

While CNNs specialized MLPs for spatial patterns, sequence models adapted neural networks for temporal dependencies. RNNs introduced the fundamental concept of maintaining and updating state, a building block that influenced how networks could process sequential information, (Elman 2002).

The development of LSTMs and GRUs brought sophisticated gating mechanisms to neural networks (Hochreiter and Schmidhuber 1997; Cho et al. 2014). These gates, themselves small MLPs, showed how simple feedforward computations could be composed to control information flow. This concept of using neural networks to modulate other neural networks became a recurring pattern in architecture design.

Perhaps most significantly, sequence models demonstrated the power of adaptive computation paths. Unlike the fixed patterns of MLPs and CNNs, RNNs showed how networks could process variable-length inputs by reusing weights over time. This insight, that architectural patterns could adapt to input structure, laid groundwork for more flexible architectures.

Sequence models also popularized the concept of attention through encoder-decoder architectures (Bahdanau, Cho, and Bengio 2014). Initially introduced as an improvement to machine translation, attention mechanisms showed how networks could learn to dynamically focus on relevant information. This building block would later become the foundation of Transformer architectures.

4.6.4 Modern Architectures: Synthesis and Innovation

Modern architectures, particularly Transformers, represent a sophisticated synthesis of these fundamental building blocks. Rather than introducing entirely new patterns, they innovate through clever combination and refinement of existing components. Consider the Transformer architecture: at its core, we find MLP-style feedforward networks processing features between attention layers. The attention mechanism itself builds on ideas from sequence models but removes the recurrent connection, instead using position embeddings[6] inspired by CNN intuitions. The architecture extensively utilizes skip connections (see Figure 4.10)[7], inherited from ResNets, while layer normalization, evolved from CNN's batch normalization, stabilizes training (Ba, Kiros, and Hinton 2016).

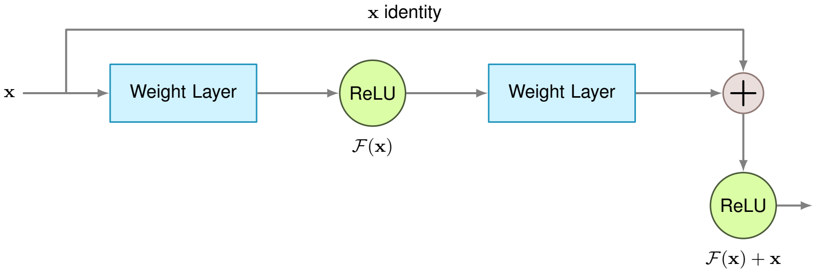


Figure 4.10: Example of a skip connection, where a computation is performed and its residual is also added to the result.

This composition of building blocks creates something greater than the sum of its parts. The self-attention mechanism, while building on previous attention concepts, enables a new form of dynamic pattern processing. The arrangement of these components, attention followed by feedforward layers, with skip connections and normalization, has proven so effective it's become a template for new architectures.

Even recent innovations in vision and language models follow this pattern of recombining fundamental building blocks. Vision Transformers adapt the Transformer architecture to images while maintaining its essential components (Dosovitskiy et al. 2021). Large language models scale up these patterns while introducing refinements like grouped-query attention or sliding window attention, yet still rely on the core building blocks established through this

[6] Position Embeddings: Vector representations that encode the position of elements within a sequence in neural network processing.

[7] Skip Connections: Connections that skip one or more layers in a neural network by feeding the output of one layer as the input to subsequent layers, enhancing gradient flow during training.

architectural evolution (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, and al. 2020).

To illustrate how these modern architectures synthesize and innovate upon previous approaches, consider the following comparison of primitive utilization across different neural network architectures:

Table 4.2: Comparison of primitive utilization across neural network architectures.

Primitive Type	MLP	CNN	RNN	Transformer
Computational	Matrix Multiplication	Convolution (Matrix Mult.)	Matrix Mult. + State Update	Matrix Mult. + Attention
Memory Access	Sequential	Strided	Sequential + Random	Random (Attention)
Data Movement	Broadcast	Sliding Window	Sequential	Broadcast + Gather

As shown in Table 4.2, Transformers combine elements from previous architectures while introducing new patterns. They retain the core matrix multiplication operations common to all architectures but introduce a more complex memory access pattern with their attention mechanism. Their data movement patterns blend the broadcast operations of MLPs with the gather operations reminiscent of more dynamic architectures.

This synthesis of primitives in Transformers exemplifies how modern architectures innovate by recombining and refining existing building blocks, rather than inventing entirely new computational paradigms. Also, this evolutionary process provides insight into the development of future architectures and helps to guide the design of efficient systems to support them.

4.7 System-Level Building Blocks

After having examined different deep learning architectures, we can distill their system requirements into fundamental primitives that underpin both hardware and software implementations. These primitives represent operations that cannot be broken down further while maintaining their essential characteristics. Just as complex molecules are built from basic atoms, sophisticated neural networks are constructed from these fundamental operations.

4.7.1 Core Computational Primitives

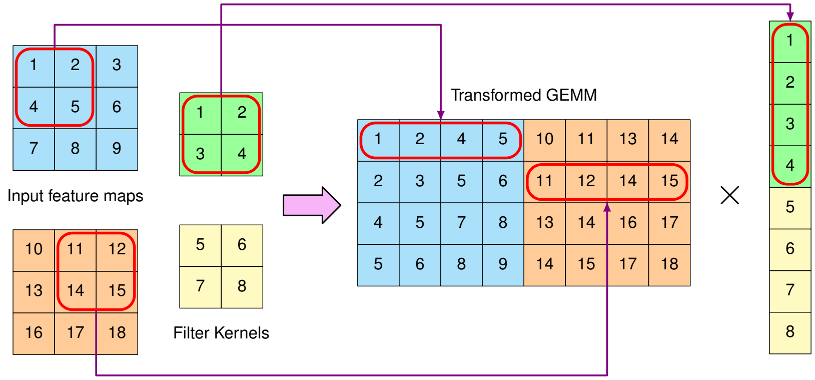
Three fundamental operations serve as the building blocks for all deep learning computations: matrix multiplication, sliding window operations[8], and dynamic computation[9]. What makes these operations primitive is that they cannot be further decomposed without losing their essential computational properties and efficiency characteristics.

Matrix multiplication represents the most basic form of transforming sets of features. When we multiply a matrix of inputs by a matrix of weights, we're computing weighted combinations, which is the fundamental operation of neural networks. For example, in our MNIST network, each 784-dimensional

[8] A technique in signal processing and computer vision where a window moves across data, computing results from subsets, essential in CNNs.

[9] Computational processes where the operations adjust based on input data, used prominently in machine learning models like the Transformer.

input vector multiplies with a 784×100 weight matrix. This pattern appears everywhere: MLPs use it directly for layer computations, CNNs reshape convolutions into matrix multiplications (turning a 3×3 convolution into a matrix operation, as illustrated in Figure 4.11), and Transformers use it extensively in their attention mechanisms.



1	2	3
4	5	6
7	8	9

10	11	12
13	14	15
16	17	18

1	2
3	4

5	6
7	8

1	2	4	5	10	11	13	14
2	3	5	6	11	12	14	15
4	5	7	8	13	14	16	17
5	6	8	9	14	15	17	18

1
2
3
4
5
6
7
8

Figure 4.11: Depiction of how `im2col` can map a convolution into a dense matrix multiplication for better efficiency.

In modern systems, matrix multiplication maps to specific hardware and software implementations. Hardware accelerators provide specialized tensor cores that can perform thousands of multiply-accumulates in parallel, NVIDIA's A100 tensor cores can achieve up to 312 TFLOPS (32-bit) through massive parallelization of these operations. Software frameworks like PyTorch and TensorFlow automatically map these high-level operations to optimized matrix libraries (NVIDIA cuBLAS, Intel MKL) that exploit these hardware capabilities.

Sliding window operations compute local relationships by applying the same operation to chunks of data. In CNNs processing MNIST images, a 3×3 convolution filter slides across the 28×28 input, requiring 26×26 windows of computation,[10] assuming a stride size of 1. Modern hardware accelerators implement this through specialized memory access patterns and data buffering schemes that optimize data reuse. For example, Google's TPU uses a 128×128 systolic array where data flows systematically through processing elements, allowing each input value to be reused across multiple computations without accessing memory. Software frameworks optimize these operations by transforming them into efficient matrix multiplications (a 3×3 convolution becomes a $9 \times N$ matrix multiplication) and carefully managing data layout in memory to maximize spatial locality.

Dynamic computation, where the operation itself depends on the input data, emerged prominently with attention mechanisms but represents a fundamental capability needed for adaptive processing. In Transformer attention, each query dynamically determines its interaction weights with all keys; for a sequence of length 512, this means 512 different weight patterns must be computed on the fly. Unlike fixed patterns where we know the computation graph in advance, dynamic computation requires runtime decisions. This creates specific implementation challenges; hardware must provide flexible routing of data (modern GPUs use dynamic scheduling) and support variable computation

[10] The 26×26 output dimension comes from the formula (N-F+1) where N is the input dimension (28) and F is the filter size (3), calculated as: 28-3+1=26 for both dimensions.

patterns, while software frameworks need efficient mechanisms for handling data-dependent execution paths (PyTorch's dynamic computation graphs, TensorFlow's dynamic control flow).

These primitives combine in sophisticated ways in modern architectures. A Transformer layer processing a sequence of 512 tokens demonstrates this clearly: it uses matrix multiplications for feature projections (512×512 operations implemented through tensor cores), may employ sliding windows for efficient attention over long sequences (using specialized memory access patterns for local regions), and requires dynamic computation for attention weights (computing 512×512 attention patterns at runtime). The way these primitives interact creates specific demands on system design, ranging from memory hierarchy organization to computation scheduling.

The building blocks we've discussed help explain why certain hardware features exist (like tensor cores for matrix multiplication) and why software frameworks organize computations in particular ways (like batching similar operations together). As we move from computational primitives to consider memory access and data movement patterns, it's important to recognize how these fundamental operations shape the demands placed on memory systems and data transfer mechanisms. The way computational primitives are implemented and combined has direct implications for how data needs to be stored, accessed, and moved within the system.

4.7.2 Memory Access Primitives

The efficiency of deep learning systems heavily depends on how they access and manage memory. In fact, memory access often becomes the primary bottleneck in modern ML systems, even though a matrix multiplication unit might be capable of performing thousands of operations per cycle, it will sit idle if data isn't available at the right time. For example, accessing data from DRAM[11] typically takes hundreds of cycles, while on-chip computation takes only a few cycles.

Three fundamental memory access patterns dominate in deep learning architectures: sequential access, strided access, and random access. Each pattern creates different demands on the memory system and offers different opportunities for optimization.

Sequential access is the simplest and most efficient pattern. Consider an MLP performing matrix multiplication with a batch of MNIST images: it needs to access both the 784×100 weight matrix and the input vectors sequentially. This pattern maps well to modern memory systems; DRAM can operate in burst mode for sequential reads (achieving up to 400 GB/s in modern GPUs), and hardware prefetchers can effectively predict and fetch upcoming data. Software frameworks optimize for this by ensuring data is laid out contiguously in memory and aligning data to cache line boundaries.

Strided access appears prominently in CNNs, where each output position needs to access a window of input values at regular intervals. For a CNN processing MNIST images with 3×3 filters, each output position requires accessing 9 input values with a stride matching the input width. While less efficient than sequential access, hardware supports this through pattern-aware

[11] DRAM: Dynamic Random Access Memory, used for main system memory.

caching strategies and specialized memory controllers. Software frameworks often transform these strided patterns into sequential access through data layout reorganization, where the im2col transformation[12] in deep learning frameworks converts convolution's strided access into efficient matrix multiplications.

Random access poses the greatest challenge for system efficiency. In a Transformer processing a sequence of 512 tokens, each attention operation potentially needs to access any position in the sequence, creating unpredictable memory access patterns. Random access can severely impact performance through cache misses (potentially causing 100+ cycle stalls per access) and unpredictable memory latencies. Systems address this through large cache hierarchies (modern GPUs have several MB of L2 cache) and sophisticated prefetching strategies, while software frameworks employ techniques like attention pattern pruning to reduce random access requirements.

These different memory access patterns contribute significantly to the overall memory requirements of each architecture. To illustrate this, Table 4.3 compares the memory complexity of MLPs, CNNs, RNNs, and Transformers.

Table 4.3: DNN architecture complexity. Note that for RNNs, parameter storage is bounded by $O(N \times h)$ when $N > h$.

Architecture	Input Dependency	Parameter Storage	Activation Storage	Scaling Behavior
MLP	Linear	$O(N \times W)$	$O(B \times W)$	Predictable
CNN	Constant	$O(K \times C)$	$O(B \times H_{\text{img}} \times W_{\text{img}})$	Efficient
RNN	Linear	$O(h^2)$	$O(B \times T \times h)$	Challenging
Transformer	Quadratic	$O(N \times d)$	$O(B \times N^2)$	Problematic

Where:

- N: Input or sequence size
- W: Layer width
- B: Batch size
- K: Kernel size
- C: Number of channels
- H_{img}: Height of input feature map (CNN)
- W_{img}: Width of input feature map (CNN)
- h: Hidden state size (RNN)
- T: Sequence length
- d: Model dimensionality

Table 4.3 reveals how memory requirements scale with different architectural choices. The quadratic scaling of activation storage in Transformers, for instance, highlights the need for large memory capacities and efficient memory management in systems designed for Transformer-based workloads. In contrast, CNNs exhibit more favorable memory scaling due to their parameter sharing and localized processing. These memory complexity considerations are crucial when making system-level design decisions, such as choosing memory hierarchy configurations and developing memory optimization strategies.

[12] im2col: An algorithm that transforms input data for efficient matrix multiplication in CNNs.

The impact of these patterns becomes clearer when we consider data reuse opportunities. In CNNs, each input pixel participates in multiple convolution windows (typically 9 times for a 3×3 filter), making effective data reuse fundamental for performance. Modern GPUs provide multi-level cache hierarchies (L1, L2, shared memory) to capture this reuse, while software techniques like loop tiling ensure data remains in cache once loaded.

Working set size, the amount of data needed simultaneously for computation, varies dramatically across architectures. An MLP layer processing MNIST images might need only a few hundred KB (weights plus activations), while a Transformer processing long sequences can require several MB just for storing attention patterns. These differences directly influence hardware design choices, like the balance between compute units and on-chip memory, and software optimizations like activation checkpointing or attention approximation techniques.

Having a good grasp of these memory access patterns is essential as architectures evolve. The shift from CNNs to Transformers, for instance, has driven the development of hardware with larger on-chip memories and more sophisticated caching strategies to handle increased working sets and more dynamic access patterns. Future architectures will likely continue to be shaped by their memory access characteristics as much as their computational requirements.

4.7.3 Data Movement Primitives

While computational and memory access patterns define what operations occur where, data movement primitives characterize how information flows through the system. These patterns are key because data movement often consumes more time and energy than computation itself, as moving data from off-chip memory typically requires 100-1000$ imes$ more energy than performing a floating-point operation.

Four fundamental data movement patterns are prevalent in deep learning architectures: broadcast, scatter, gather, and reduction. Figure 4.12 illustrates these patterns and their relationships. Broadcast operations send the same data to multiple destinations simultaneously. In matrix multiplication with batch size 32, each weight must be broadcast to process different inputs in parallel. Modern hardware supports this through specialized interconnects, NVIDIA GPUs provide hardware multicast capabilities, achieving up to 600 GB/s broadcast bandwidth, while TPUs use dedicated broadcast buses. Software frameworks optimize broadcasts by restructuring computations (like matrix tiling) to maximize data reuse.

Scatter operations distribute different elements to different destinations. When parallelizing a 512×512 matrix multiplication across GPU cores, each core receives a subset of the computation. This parallelization is important for performance but challenging, as memory conflicts and load imbalance, can reduce efficiency by 50% or more. Hardware provides flexible interconnects (like NVIDIA's NVLink offering 600 GB/s bi-directional bandwidth), while software frameworks employ sophisticated work distribution algorithms to maintain high utilization.

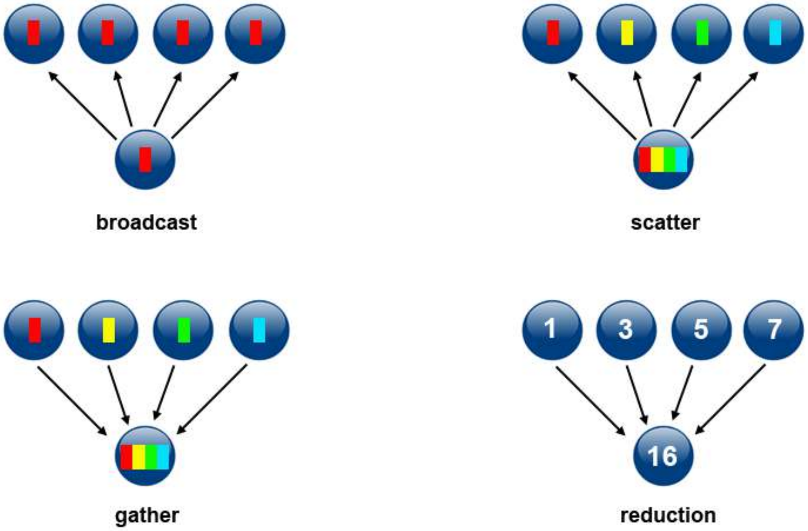


Figure 4.12: Collective communication routines.

Gather operations collect data from multiple sources. In Transformer attention with sequence length 512, each query must gather information from 512 different key-value pairs. These irregular access patterns are challenging, random gathering can be 10× slower than sequential access. Hardware supports this through high-bandwidth interconnects and large caches, while software frameworks employ techniques like attention pattern pruning to reduce gathering overhead.

Reduction operations combine multiple values into a single result through operations like summation. When computing attention scores in Transformers or layer outputs in MLPs, efficient reduction is essential. Hardware implements tree-structured reduction networks (reducing latency from $O(n)$ to $O(\log n)$), while software frameworks use optimized parallel reduction algorithms that can achieve near-theoretical peak performance.

These patterns combine in sophisticated ways. A Transformer attention operation with sequence length 512 and batch size 32 involves:

- Broadcasting query vectors (512×64 elements)
- Gathering relevant keys and values ($512 \times 512 \times 64$ elements)
- Reducing attention scores (512×512 elements per sequence)

The evolution from CNNs to Transformers has increased reliance on gather and reduction operations, driving hardware innovations like more flexible interconnects and larger on-chip memories. As models grow (some now exceeding 100 billion parameters), efficient data movement becomes increasingly critical, leading to innovations like near-memory processing and sophisticated data flow optimizations.

4.7.4 System Design Impact

The computational, memory access, and data movement primitives we've explored form the foundational requirements that shape the design of systems for deep learning. The way these primitives influence hardware design, create common bottlenecks, and drive trade-offs is important for developing efficient and effective deep learning systems.

One of the most significant impacts of these primitives on system design is the push towards specialized hardware. The prevalence of matrix multiplications and convolutions in deep learning has led to the development of tensor processing units (TPUs) and tensor cores in GPUs, which are specifically designed to perform these operations efficiently. These specialized units can perform many multiply-accumulate operations in parallel, dramatically accelerating the core computations of neural networks.

Memory systems have also been profoundly influenced by the demands of deep learning primitives. The need to support both sequential and random access patterns efficiently has driven the development of sophisticated memory hierarchies. High-bandwidth memory (HBM)[13] has become common in AI accelerators to support the massive data movement requirements, especially for operations like attention mechanisms in Transformers. On-chip memory hierarchies have grown in complexity, with multiple levels of caching and scratchpad memories to support the diverse working set sizes of different neural network layers.

The data movement primitives have particularly influenced the design of interconnects and on-chip networks. The need to support efficient broadcasts, gathers, and reductions has led to the development of more flexible and higher-bandwidth interconnects. Some AI chips now feature specialized networks-on-chip designed to accelerate common data movement patterns in neural networks.

Table 4.4 summarizes the system implications of these primitives:

Table 4.4: System implications of primitives.

Primitive	Hardware Impact	Software Optimization	Key Challenges
Matrix Multiplication	Tensor Cores	Batching, GEMM libraries	Parallelization, precision
Sliding Window	Specialized datapaths	Data layout optimization	Stride handling
Dynamic Computation	Flexible routing	Dynamic graph execution	Load balancing
Sequential Access	Burst mode DRAM	Contiguous allocation	Access latency
Random Access	Large caches	Memory-aware scheduling	Cache misses
Broadcast	Specialized interconnects	Operation fusion	Bandwidth
Gather/Scatter	High-bandwidth memory	Work distribution	Load balancing

Despite these advancements, several common bottlenecks persist in deep learning systems. Memory bandwidth often remains a key limitation, particularly for models with large working sets or those that require frequent random access. The energy cost of data movement, especially between off-chip memory and processing units, continues to be a significant concern. For large-scale

[13] High-bandwidth memory (HBM): A type of stacked DRAM designed to provide high-speed data access for processing units.

models, the communication overhead in distributed training can become a bottleneck, limiting scaling efficiency.

System designers must navigate complex trade-offs in supporting different primitives, each with unique characteristics that influence system design and performance. For example, optimizing for the dense matrix operations common in MLPs and CNNs might come at the cost of flexibility needed for the more dynamic computations in attention mechanisms. Supporting large working sets for Transformers might require sacrificing energy efficiency.

Balancing these trade-offs requires careful consideration of the target workloads and deployment scenarios. Having a good grip on the nature of each primitive guides the development of both hardware and software optimizations in deep learning systems, allowing designers to make informed decisions about system architecture and resource allocation.

4.8 Conclusion

Deep learning architectures, despite their diversity, exhibit common patterns in their algorithmic structures that significantly influence computational requirements and system design. In this chapter, we explored the intricate relationship between high-level architectural concepts and their practical implementation in computing systems.

From the straightforward dense connections of MLPs to the complex, dynamic patterns of Transformers, each architecture builds upon a set of fundamental building blocks. These core computational primitives, including matrix multiplication, sliding windows, and dynamic computation, recur across various architectures, forming a universal language of deep learning computation.

The identification of these shared elements provides a valuable framework for understanding and designing deep learning systems. Each primitive brings its own set of requirements in terms of memory access patterns and data movement, which in turn shape both hardware and software design decisions. This relationship between algorithmic intent and system implementation is crucial for optimizing performance and efficiency.

As the field of deep learning continues to evolve, the ability to efficiently support and optimize these fundamental building blocks will be key to the development of more powerful and scalable systems. Future advancements in deep learning are likely to stem not only from novel architectural designs but also from innovative approaches to implementing and optimizing these essential computational patterns.

In conclusion, understanding the mapping between neural architectures and their computational requirements is vital for pushing the boundaries of what's possible in artificial intelligence. As we look to the future, the interplay between algorithmic innovation and systems optimization will continue to drive progress in this rapidly advancing field.

4.9 Resources

ℹ Slides

- *Coming soon.*

! Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 5

AI Workflow

Figure 5.1: *DALL·E 3 Prompt: Create a rectangular illustration of a stylized flowchart representing the AI workflow/pipeline. From left to right, depict the stages as follows: 'Data Collection' with a database icon, 'Data Preprocessing' with a filter icon, 'Model Design' with a brain icon, 'Training' with a weight icon, 'Evaluation' with a checkmark, and 'Deployment' with a rocket. Connect each stage with arrows to guide the viewer horizontally through the AI processes, emphasizing these steps' sequential and interconnected nature.*

Purpose

What are the diverse elements of AI systems and how do we combine to create effective machine learning system solutions?

The creation of practical AI solutions requires the orchestration of multiple components into coherent workflows. Workflow design highlights the connections and interactions that animate these components. This systematic perspective reveals how data flow, model training, and deployment considerations are intertwined to form robust AI systems. Analyzing these interconnections offers important insights into system-level design choices, establishing a framework for understanding how theoretical concepts can be translated into deployable solutions that meet real-world needs.

Learning Objectives

- Understand the ML lifecycle and gain insights into the structured approach and stages of developing, deploying, and maintaining machine learning models.
- Identify the unique challenges and distinctions between lifecycles for traditional machine learning and specialized applications.
- Explore the various people and roles involved in ML projects.
- Examine the importance of system-level considerations, including resource constraints, infrastructure, and deployment environments.
- Appreciate the iterative nature of ML lifecycles and how feedback loops drive continuous improvement in real-world applications.

5.1 Overview

The machine learning lifecycle is a systematic, interconnected process that guides the transformation of raw data into actionable models deployed in real-world applications. Each stage builds upon the outcomes of the previous one, creating an iterative cycle of refinement and improvement that supports robust, scalable, and reliable systems.

Figure 5.2 illustrates the lifecycle as a series of stages connected through continuous feedback loops. The process begins with data collection, which ensures a steady input of raw data from various sources. The collected data progresses to data ingestion, where it is prepared for downstream machine learning applications. Subsequently, data analysis and curation involve inspecting and selecting the most appropriate data for the task at hand. Following this, data labeling and data validation, which nowadays involves both humans and AI itself, ensure that the data is properly annotated and verified for usability before advancing further.

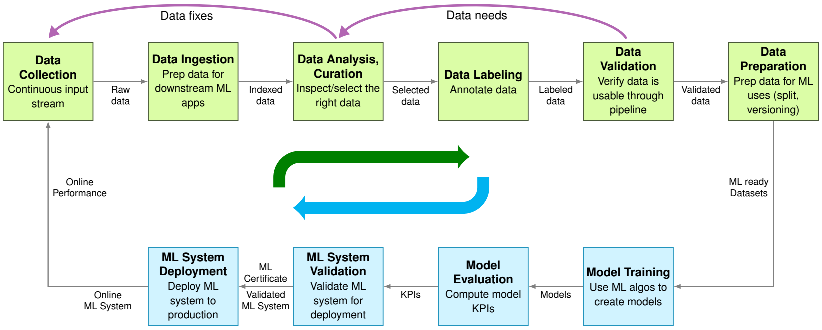


Figure 5.2: The ML lifecycle.

The data then enters the preparation stage, where it is transformed into machine learning-ready datasets through processes such as splitting and versioning. These datasets are used in the model training stage, where machine learning algorithms are applied to create predictive models. The resulting models are rigorously tested in the model evaluation stage, where performance metrics, such as key performance indicators (KPIs), are computed to assess reliability and effectiveness. The validated models move to the ML system validation phase, where they are verified for deployment readiness. Once validated, these models are integrated into production systems during the ML system deployment stage, ensuring alignment with operational requirements. The final stage tracks the performance of deployed systems in real time, enabling continuous adaptation to new data and evolving conditions.

This general lifecycle forms the backbone of machine learning systems, with each stage contributing to the creation, validation, and maintenance of scalable and efficient solutions. While the lifecycle provides a detailed view of the interconnected processes in machine learning systems, it can be distilled into a simplified framework for practical implementation.

Each stage aligns with one of the following overarching categories:

- **Data Collection and Preparation** ensures the availability of high-quality, representative datasets.
- **Model Development and Training** focuses on creating accurate and efficient models tailored to the problem at hand.
- **Evaluation and Validation** rigorously tests models to ensure reliability and robustness in real-world conditions.
- **Deployment and Integration** translates models into production-ready systems that align with operational realities.
- **Monitoring and Maintenance** ensures ongoing system performance and adaptability in dynamic environments.

A defining feature of this framework is its iterative and dynamic nature. Feedback loops, such as those derived from monitoring that guide data collection improvements or deployment adjustments, ensure that machine learning systems maintain effectiveness and relevance over time. This adaptability is critical for addressing challenges such as shifting data distributions, operational constraints, and evolving user requirements.

By studying this framework, we establish a solid foundation for exploring specialized topics such as data engineering, model optimization, and deployment strategies in subsequent chapters. Viewing the ML lifecycle as an integrated and iterative process promotes a deeper understanding of how systems are designed, implemented, and maintained over time. To that end, this chapter focuses on the machine learning lifecycle as a systems-level framework, providing a high-level overview that bridges theoretical concepts with practical implementation. Through an examination of the lifecycle in its entirety, we gain insight into the interdependencies among its stages and the iterative processes that ensure long-term system scalability and relevance.

5.1.1 Definition

The machine learning (ML) lifecycle is a structured, iterative process that guides the development, evaluation, and continual improvement of machine learning systems. Integrating ML into broader software engineering practices introduces unique challenges that necessitate systematic approaches to experimentation, evaluation, and adaptation over time (Amershi et al. 2019).

ℹ Definition of the Machine Learning Lifecycle

The Machine Learning (ML) Lifecycle is a *structured, iterative process* that defines the *key stages* involved in the *development, deployment, and refinement* of ML systems. It encompasses *interconnected steps* such as *problem formulation, data collection, model training, evaluation, deployment, and monitoring*. The lifecycle emphasizes *feedback loops and continuous improvement*, ensuring that systems remain *robust, scalable, and responsive* to *changing requirements and real-world conditions*.

Rather than prescribing a fixed methodology, the ML lifecycle focuses on achieving specific objectives at each stage. This flexibility allows practitioners to adapt the process to the unique constraints and goals of individual projects. Typical stages include problem formulation, data acquisition and preprocessing, model development and training, evaluation, deployment, and ongoing optimization.

Although these stages may appear sequential, they are frequently revisited, creating a dynamic and interconnected process. The iterative nature of the lifecycle encourages feedback loops, whereby insights from later stages, including deployment, can inform earlier phases, including data preparation or model architecture design. This adaptability is essential for managing the uncertainties and complexities inherent in real-world ML applications.

From an instructional standpoint, the ML lifecycle provides a clear framework for organizing the study of machine learning systems. By decomposing the field into well-defined stages, students can engage more systematically with its core components. This structure mirrors industrial practice while supporting deeper conceptual understanding.

It is important to distinguish between the ML lifecycle and machine learning operations (MLOps), as the two are often conflated. The ML lifecycle, as presented in this chapter, emphasizes the stages and evolution of ML systems—the "what" and "why" of system development. In contrast, MLOps, which will be discussed in the MLOps Chapter, addresses the "how," focusing on tools, processes, and automation that support efficient implementation and maintenance. Introducing the lifecycle first provides a conceptual foundation for understanding the operational aspects that follow.

5.1.2 Traditional vs. AI Lifecycles

Software development lifecycles have evolved through decades of engineering practice, establishing well-defined patterns for system development. Traditional lifecycles consist of sequential phases: requirements gathering, system design, implementation, testing, and deployment. Each phase produces specific artifacts that serve as inputs to subsequent phases. In financial software development, for instance, the requirements phase produces detailed specifications for transaction processing, security protocols, and regulatory compliance—specifications that directly translate into system behavior through explicit programming.

Machine learning systems require a fundamentally different approach to this traditional lifecycle model. The deterministic nature of conventional software, where behavior is explicitly programmed, contrasts sharply with the probabilistic nature of ML systems. Consider financial transaction processing: traditional systems follow predetermined rules (if account balance > transaction amount, then allow transaction), while ML-based fraud detection systems learn to recognize suspicious patterns from historical transaction data. This shift from explicit programming to learned behavior fundamentally reshapes the development lifecycle.

The unique characteristics of machine learning systems, characterized by data dependency, probabilistic outputs, and evolving performance, introduce new dynamics that alter how lifecycle stages interact. These systems require ongoing refinement, with insights from later stages frequently feeding back into earlier ones. Unlike traditional systems, where lifecycle stages aim to produce stable outputs, machine learning systems are inherently dynamic and must adapt to changing data distributions and objectives.

The key distinctions are summarized in Table 5.1 below:

Table 5.1: Differences between traditional and ML lifecycles.

Aspect	Traditional Software Lifecycles	Machine Learning Lifecycles
Problem Definition	Precise functional specifications are defined upfront.	Performance-driven objectives evolve as the problem space is explored.
Development Process	Linear progression of feature implementation.	Iterative experimentation with data, features and models.
Testing and Validation	Deterministic, binary pass/fail testing criteria.	Statistical validation and metrics that involve uncertainty.
Deployment	Behavior remains static until explicitly updated.	Performance may change over time due to shifts in data distributions.
Maintenance	Maintenance involves modifying code to address bugs or add features.	Continuous monitoring, updating data pipelines, retraining models, and adapting to new data distributions.
Feedback Loops	Minimal; later stages rarely impact earlier phases.	Frequent; insights from deployment and monitoring often refine earlier stages like data preparation and model design.

These differences underline the need for a robust ML lifecycle framework that can accommodate iterative development, dynamic behavior, and data-driven decision-making. This lifecycle ensures that machine learning systems remain effective not only at launch but throughout their operational lifespan, even as environments evolve.

5.2 Lifecycle Stages

The AI lifecycle consists of several interconnected stages, each essential to the development and maintenance of effective machine learning systems. While the specific implementation details may vary across projects and organizations, Figure 5.3 provides a high-level illustration of the ML system development

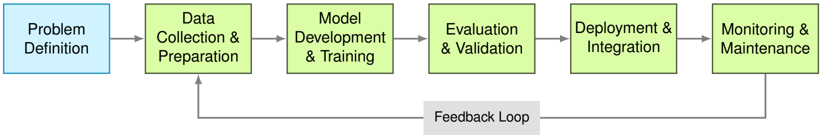


Figure 5.3: ML lifecycle overview.

lifecycle. This chapter focuses on the overview, with subsequent chapters diving into the implementation aspects of each stage.

Problem Definition and Requirements: The first stage involves clearly defining the problem to be solved, establishing measurable performance objectives, and identifying key constraints. Precise problem definition ensures alignment between the system's goals and the desired outcomes.

Data Collection and Preparation: This stage includes gathering relevant data, cleaning it, and preparing it for model training. This process often involves curating diverse datasets, ensuring high-quality labeling, and developing preprocessing pipelines to address variations in the data.

Model Development and Training: In this stage, researchers select appropriate algorithms, design model architectures, and train models using the prepared data. Success depends on choosing techniques suited to the problem and iterating on the model design for optimal performance.

Evaluation and Validation: Evaluation involves rigorously testing the model's performance against predefined metrics and validating its behavior in different scenarios. This stage ensures the model is not only accurate but also reliable and robust in real-world conditions.

Deployment and Integration: Once validated, the trained model is integrated into production systems and workflows. This stage requires addressing practical challenges such as system compatibility, scalability, and operational constraints.

Monitoring and Maintenance: The final stage focuses on continuously monitoring the system's performance in real-world environments and maintaining or updating it as necessary. Effective monitoring ensures the system remains relevant and accurate over time, adapting to changes in data, requirements, or external conditions.

A Case Study in Medical AI: To further ground our discussion on these stages, we will explore Google's Diabetic Retinopathy (DR) screening project as a case study. This project exemplifies the transformative potential of machine learning in medical imaging analysis, an area where the synergy between algorithmic innovation and robust systems engineering plays a pivotal role. Building upon the foundational work by Gulshan et al. (2016), which demonstrated the effectiveness of deep learning algorithms in detecting diabetic retinopathy from retinal fundus photographs, the project progressed from research to real-world deployment, revealing the complex challenges that characterize modern ML systems.

Diabetic retinopathy, a leading cause of preventable blindness worldwide, can be detected through regular screening of retinal photographs. Figure 5.4 illustrates examples of such images: (A) a healthy retina and (B) a retina with diabetic retinopathy, marked by hemorrhages (red spots). The goal is to train a model to detect the hemorrhages.

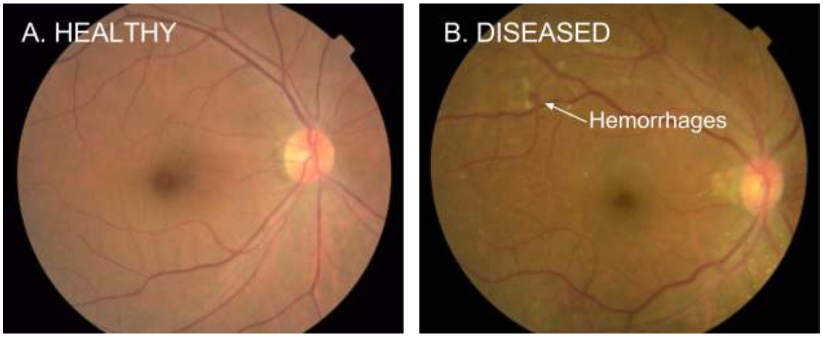


Figure 5.4: Retinal fundus photos: (A) healthy retina and (B) retina with diabetic retinopathy showing hemorrhages (red spots). Source: Google

On the surface, the goal appears straightforward: develop an AI system that could analyze retinal images and identify signs of DR with accuracy comparable to expert ophthalmologists. However, as the project progressed from research to real-world deployment, it revealed the complex challenges that characterize modern ML systems.

The initial results in controlled settings were promising. The system achieved performance comparable to expert ophthalmologists in detecting DR from high-quality retinal photographs. Yet, when the team attempted to deploy the system in rural clinics across Thailand and India, they encountered a series of challenges that spanned the entire ML lifecycle, from data collection through deployment and maintenance.

This case study will serve as a recurring thread throughout this chapter to illustrate how success in machine learning systems depends on more than just model accuracy. It requires careful orchestration of data pipelines, training infrastructure, deployment systems, and monitoring frameworks. Furthermore, the project highlights the iterative nature of ML system development, where real-world deployment often necessitates revisiting and refining earlier stages.

While this narrative is inspired by Google's documented experiences in Thailand and India, certain aspects have been embellished to emphasize specific challenges frequently encountered in real-world healthcare ML deployments. These enhancements are to provide a richer understanding of the complexities involved while maintaining credibility and relevance to practical applications.

5.3 Problem Definition

The development of machine learning systems begins with a critical challenge that fundamentally differs from traditional software development: defining not just what the system should do, but how it should learn to do it. Unlike conventional software, where requirements directly translate into implementation rules, ML systems require teams to consider how the system will learn from data while operating within real-world constraints. This stage lays the foundation for all subsequent phases in the ML lifecycle.

In our case study, diabetic retinopathy is a problem that blends technical complexity with global healthcare implications. With 415 million diabetic patients at risk of blindness worldwide and limited access to specialists in underserved regions, defining the problem required balancing technical goals, such as expert-level diagnostic accuracy, with practical constraints. The system needed to prioritize cases for early intervention while operating effectively in resource-limited settings. These constraints showcased how problem definition must integrate learning capabilities with operational needs to deliver actionable and sustainable solutions.

5.3.1 Requirements and System Impact

Defining an ML problem involves more than specifying desired performance metrics. It requires a deep understanding of the broader context in which the system will operate. For instance, developing a system to detect DR with expert-level accuracy might initially appear to be a straightforward classification task. After all, one might assume that training a model on a sufficiently large dataset of labeled retinal images and evaluating its performance against standard metrics would suffice.

However, real-world challenges complicate this picture. ML systems must function effectively in diverse environments, where factors like computational constraints, data variability, and integration requirements play significant roles. For example, the DR system needed to detect subtle features like microaneurysms[0], hemorrhages[1], and hard exudates[2] across retinal images of varying quality while operating within the limitations of hardware in rural clinics. A model that performs well in isolation may falter if it cannot handle operational realities, such as inconsistent imaging conditions or time-sensitive clinical workflows. Addressing these factors requires aligning learning objectives with system constraints, ensuring the system's long-term viability in its intended context.

5.3.2 Definition Workflow

Establishing clear and actionable problem definitions involves a multi-step workflow that bridges technical, operational, and user considerations. The process begins with identifying the core objective of the system—what tasks it must perform and what constraints it must satisfy. Teams collaborate with stakeholders to gather domain knowledge, outline requirements, and anticipate challenges that may arise in real-world deployment.

In the DR project, this phase involved close collaboration with clinicians to determine the diagnostic needs of rural clinics. Key decisions, such as balancing model complexity with hardware limitations and ensuring interpretability for healthcare providers, were made during this phase. The team's iterative approach also accounted for regulatory considerations, such as patient privacy and compliance with healthcare standards. This collaborative process ensured that the problem definition aligned with both technical feasibility and clinical relevance.

5.3.3 Scale and Distribution

As ML systems scale, their problem definitions must adapt to new operational challenges. For example, the DR project initially focused on a limited number

[0] Microaneurysms: Small bulges in blood vessels of the retina commonly seen in diabetic retinopathy.

[1] Hemorrhages: Blood that has leaked from the blood vessels into the surrounding tissues.

[2] Hard Exudates: Deposits of lipids or fats indicative of leakage from impaired retinal blood vessels.

of clinics with consistent imaging setups. However, as the system expanded to include clinics with varying equipment, staff expertise, and patient demographics, the original problem definition required adjustments to accommodate these variations.

Scaling also introduces data challenges. Larger datasets may include more diverse edge cases, which can expose weaknesses in the initial model design. In the DR project, for instance, expanding the deployment to new regions introduced variations in imaging equipment and patient populations that required further tuning of the system. Defining a problem that accommodates such diversity from the outset ensures the system can handle future expansion without requiring a complete redesign.

5.3.4 Systems Thinking

Problem definition, viewed through a systems lens, connects deeply with every stage of the ML lifecycle. Choices made during this phase shape how data is collected, how models are developed, and how systems are deployed and maintained. A poorly defined problem can lead to inefficiencies or failures in later stages, emphasizing the need for a holistic perspective.

Feedback loops are central to effective problem definition. As the system evolves, real-world feedback from deployment and monitoring often reveals new constraints or requirements that necessitate revisiting the problem definition. For example, feedback from clinicians about system usability or patient outcomes may guide refinements in the original goals. In the DR project, the need for interpretable outputs that clinicians could trust and act upon influenced both model development and deployment strategies.

Emergent behaviors[3] also play a role. A system that was initially designed to detect retinopathy might reveal additional use cases, such as identifying other conditions like diabetic macular edema, which can reshape the problem's scope and requirements. In the DR project, insights from deployment highlighted potential extensions to other imaging modalities, such as 3D Optical Coherence Tomography (OCT)[4].

Resource dependencies further highlight the interconnectedness of problem definition. Decisions about model complexity, for instance, directly affect infrastructure needs, data collection strategies, and deployment feasibility. Balancing these dependencies requires careful planning during the problem definition phase, ensuring that early decisions do not create bottlenecks in later stages.

5.3.5 Lifecycle Implications

The problem definition phase is foundational, influencing every subsequent stage of the lifecycle. A well-defined problem ensures that data collection focuses on the most relevant features, that models are developed with the right constraints in mind, and that deployment strategies align with operational realities.

In the DR project, defining the problem with scalability and adaptability in mind enabled the team to anticipate future challenges, such as accommodating new imaging devices or expanding to additional clinics. For instance, early

[3] Emergent Behavior: Unexpected phenomena or behaviors not foreseen by designers, arising from the interaction of system components.

[4] 3D Optical Coherence Tomography (OCT): A non-invasive imaging technique used to obtain high resolution cross-sectional images of the retina.

considerations of diverse imaging conditions and patient demographics reduced the need for costly redesigns later in the lifecycle. This forward-thinking approach ensured the system's long-term success and adaptability in dynamic healthcare environments.

By embedding lifecycle thinking into problem definition, teams can create systems that not only meet initial requirements but also adapt and evolve in response to changing conditions. This ensures that ML systems remain effective, scalable, and impactful over time.

5.4 Data Collection

Data is the foundation of machine learning systems, yet collecting and preparing data for ML applications introduces challenges that extend far beyond gathering enough training examples. Modern ML systems often need to handle terabytes of data, which range from raw, unstructured inputs to carefully annotated datasets, while maintaining quality, diversity, and relevance for model training. For medical systems like DR screening, data preparation must meet the highest standards to ensure diagnostic accuracy.

In the DR project, data collection involved a development dataset of 128,000 retinal fundus photographs evaluated by a panel of 54 ophthalmologists, with each image reviewed by 3-7 experts. This collaborative effort ensured high-quality labels that captured clinically relevant features like microaneurysms, hemorrhages, and hard exudates. Additionally, clinical validation datasets comprising 12,000 images provided an independent benchmark to test the model's robustness against real-world variability, illustrating the importance of rigorous and representative data collection. The scale and complexity of this effort highlight how domain expertise and interdisciplinary collaboration are critical to building datasets for high-stakes ML systems.

5.4.1 Data Requirements and Impact

The requirements for data collection and preparation emerge from the dual perspectives of machine learning and operational constraints. In the DR project, high-quality retinal images annotated by experts were a foundational need to train accurate models. However, real-world conditions quickly revealed additional complexities. Images were collected from rural clinics using different camera equipment, operated by staff with varying levels of expertise, and often under conditions of limited network connectivity.

These operational realities shaped the system architecture in significant ways. The volume and size of high-resolution images necessitated local storage and preprocessing capabilities at clinics, as centralizing all data collection was impractical due to unreliable internet access. Furthermore, patient privacy regulations required secure data handling at every stage, from image capture to model training. Coordinating expert annotations also introduced logistical challenges, necessitating systems that could bridge the physical distance between clinics and ophthalmologists while maintaining workflow efficiency.

These considerations demonstrate how data collection requirements influence the entire ML lifecycle. Infrastructure design, annotation pipelines, and privacy protocols all play critical roles in ensuring that collected data aligns with both technical and operational goals.

5.4.2 Data Infrastructure

The flow of data through the system highlights critical infrastructure requirements at every stage. In the DR project, the journey of a single retinal image offers a glimpse into these complexities. From its capture on a retinal camera, where image quality is paramount, the data moves through local clinic systems for initial storage and preprocessing. Eventually, it must reach central systems where it is aggregated with data from other clinics for model training and validation.

At each step, the system must balance local needs with centralized aggregation requirements. Clinics with reliable high-speed internet could transmit data in real-time, but many rural locations relied on store-and-forward systems, where data was queued locally and transmitted in bulk when connectivity permitted. These differences necessitated flexible infrastructure that could adapt to varying conditions while maintaining data consistency and integrity across the lifecycle. This adaptability ensured that the system could function reliably despite the diverse operational environments of the clinics.

5.4.3 Scale and Distribution

As ML systems scale, the challenges of data collection grow exponentially. In the DR project, scaling from an initial few clinics to a broader network introduced significant variability in equipment, workflows, and operating conditions. Each clinic effectively became an independent data node, yet the system needed to ensure consistent performance and reliability across all locations.

This scaling effort also brought increasing data volumes, as higher-resolution imaging devices became standard, generating larger and more detailed images. These advances amplified the demands on storage and processing infrastructure, requiring optimizations to maintain efficiency without compromising quality. Differences in patient demographics, clinic workflows, and connectivity patterns further underscored the need for robust design to handle these variations gracefully.

Scaling challenges highlight how decisions made during the data collection phase ripple through the lifecycle, impacting subsequent stages like model development, deployment, and monitoring. For instance, accommodating higher-resolution data during collection directly influences computational requirements for training and inference, emphasizing the need for lifecycle thinking[5] even at this early stage.

5.4.4 Data Validation

Quality assurance is an integral part of the data collection process, ensuring that data meets the requirements for downstream stages. In the DR project, automated checks at the point of collection flagged issues like poor focus or incorrect framing, allowing clinic staff to address problems immediately. These

[5] Lifecycle Thinking: Considering all phases of a system's life from design to decommissioning to optimize overall performance.

proactive measures ensured that low-quality data was not propagated through the pipeline.

Validation systems extended these efforts by verifying not just image quality but also proper labeling, patient association, and compliance with privacy regulations. Operating at both local and centralized levels, these systems ensured data reliability and robustness, safeguarding the integrity of the entire ML pipeline.

5.4.5 Systems Thinking

Viewing data collection and preparation through a lifecycle lens reveals the interconnected nature of these processes. Each decision made during this phase influences subsequent stages of the ML system. For instance, choices about camera equipment and image preprocessing affect not only the quality of the training dataset but also the computational requirements for model development and the accuracy of predictions during deployment.

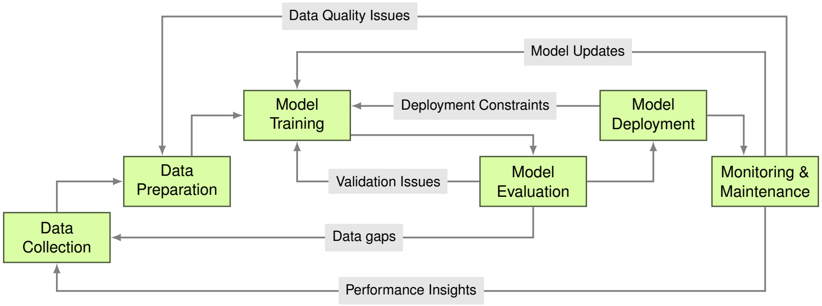


Figure 5.5: Feedback loops and dependencies between stages in the ML lifecycle.

Figure 5.5 illustrates the key feedback loops that characterize the ML lifecycle, with particular relevance to data collection and preparation. Looking at the left side of the diagram, we see how monitoring and maintenance activities feed back to both data collection and preparation stages. For example, when monitoring reveals data quality issues in production (shown by the “Data Quality Issues” feedback arrow), this triggers refinements in our data preparation pipelines. Similarly, performance insights from deployment might highlight gaps in our training data distribution (indicated by the “Performance Insights” loop back to data collection), prompting the collection of additional data to cover underrepresented cases. In the DR project, this manifested when monitoring revealed that certain demographic groups were underrepresented in the training data, leading to targeted data collection efforts to improve model fairness and accuracy across all populations.

Feedback loops are another critical aspect of this lifecycle perspective. Insights from model performance often lead to adjustments in data collection strategies, creating an iterative improvement process. For example, in the DR project, patterns observed during model evaluation influenced updates to preprocessing pipelines, ensuring that new data aligned with the system’s evolving requirements.

The scaling of data collection introduces emergent behaviors that must be managed holistically. While individual clinics may function well in isolation, the simultaneous operation of multiple clinics can lead to system-wide patterns like network congestion or storage bottlenecks. These behaviors reinforce the importance of considering data collection as a system-level challenge rather than a discrete, isolated task.

In the following chapters, we will step through each of the major stages of the lifecycle shown in Figure 5.5. We will consider several key questions like what influences data source selection, how feedback loops can be systematically incorporated, and how emergent behaviors can be anticipated and managed holistically.

In addition, by adopting a systems thinking approach, we emphasize the iterative and interconnected nature of the ML lifecycle. How do choices in data collection and preparation ripple through the entire pipeline? What mechanisms ensure that monitoring insights and performance evaluations effectively inform improvements at earlier stages? And how can governance frameworks and infrastructure design evolve to meet the challenges of scaling while maintaining fairness and efficiency? These questions will guide our exploration of the lifecycle, offering a foundation for designing robust and adaptive ML systems.

5.4.6 Lifecycle Implications

The success of ML systems depends on how effectively data collection integrates with the entire lifecycle. Decisions made in this stage affect not only the quality of the initial model but also the system's ability to evolve and adapt. For instance, data distribution shifts or changes in imaging equipment over time require the system to handle new inputs without compromising performance.

In the DR project, embedding lifecycle thinking into data management strategies ensured the system remained robust and scalable as it expanded to new clinics and regions. By proactively addressing variability and quality during data collection, the team minimized the need for costly downstream adjustments, aligning the system with long-term goals and operational realities.

5.5 Model Development

Model development and training form the core of machine learning systems, yet this stage presents unique challenges that extend far beyond selecting algorithms and tuning hyperparameters. It involves designing architectures suited to the problem, optimizing for computational efficiency, and iterating on models to balance performance with deployability. In high-stakes domains like healthcare, the stakes are particularly high, as every design decision impacts clinical outcomes.

For DR detection, the model needed to achieve expert-level accuracy while handling the high resolution and variability of retinal images. Using a deep neural network trained on their meticulously labeled dataset, the team achieved an F-score of 0.95, slightly exceeding the median score of the consulted ophthalmologists (0.91). This outcome highlights the effectiveness of state-of-the-art

[6] Transfer Learning: A method where a model developed for a task is reused as the starting point for a model on a second task.

methods, such as transfer learning[6], and the importance of interdisciplinary collaboration between data scientists and medical experts to refine features and interpret model outputs.

5.5.1 Model Requirements and Impact

The requirements for model development emerge not only from the specific learning task but also from broader system constraints. In the DR project, the model needed high sensitivity and specificity to detect different stages of retinopathy. However, achieving this purely from an ML perspective was not sufficient. The system had to meet operational constraints, including running on limited hardware in rural clinics, producing results quickly enough to fit into clinical workflows, and being interpretable enough for healthcare providers to trust its outputs.

These requirements shaped decisions during model development. While state-of-the-art accuracy might favor the largest and most complex models, such approaches were infeasible given hardware and workflow constraints. The team focused on designing architectures that balanced accuracy with efficiency, exploring lightweight models that could perform well on constrained devices. For example, techniques like pruning and quantization were employed to optimize the models for resource-limited environments, ensuring compatibility with rural clinic infrastructure.

This balancing act influenced every part of the system lifecycle. Decisions about model architecture affected data preprocessing, shaped the training infrastructure, and determined deployment strategies. For example, choosing to use an ensemble of smaller models instead of a single large model altered data batching during training, required changes to inference pipelines, and introduced complexities in how model updates were managed in production.

5.5.2 Development Workflow

The model development workflow reflects the complex interplay between data, compute resources, and human expertise. In the DR project, this process began with data exploration and feature engineering, where data scientists collaborated with ophthalmologists to identify image characteristics indicative of retinopathy.

This initial stage required tools capable of handling large medical images and facilitating experimentation with preprocessing techniques. The team needed an environment that supported collaboration, visualization, and rapid iteration while managing the sheer scale of high-resolution data.

As the project advanced to model design and training, computational demands escalated. Training deep learning models on high-resolution images required extensive GPU resources and sophisticated infrastructure. The team implemented distributed training systems that could scale across multiple machines while managing large datasets, tracking experiments, and ensuring reproducibility. These systems also supported experiment comparison, enabling rapid evaluation of different architectures, hyperparameters, and preprocessing pipelines.

Model development was inherently iterative, with each cycle, involving adjustments to DNN architectures, refinements of hyperparameters, or incorporations of new data, producing extensive metadata, including checkpoints, validation results, and performance metrics. Managing this information across the team required robust tools for experiment tracking and version control to ensure that progress remained organized and reproducible.

5.5.3 Scale and Distribution

As ML systems scale in both data volume and model complexity, the challenges of model development grow exponentially. The DR project's evolution from prototype models to production-ready systems highlights these hurdles. Expanding datasets, more sophisticated models, and concurrent experiments demanded increasingly powerful computational resources and meticulous organization.

Distributed training became essential to meet these demands. While it significantly reduced training time, it introduced complexities in data synchronization, gradient aggregation, and fault tolerance. The team relied on advanced frameworks to optimize GPU clusters, manage network latency, and address hardware failures, ensuring training processes remained efficient and reliable. These frameworks included automated failure recovery mechanisms, which helped maintain progress even in the event of hardware interruptions.

The need for continuous experimentation and improvement compounded these challenges. Over time, the team managed an expanding repository of model versions, training datasets, and experimental results. This growth required scalable systems for tracking experiments, versioning models, and analyzing results to maintain consistency and focus across the project.

5.5.4 Systems Thinking

Approaching model development through a systems perspective reveals its connections to every other stage of the ML lifecycle. Decisions about model architecture ripple through the system, influencing preprocessing requirements, deployment strategies, and clinical workflows. For instance, adopting a complex model might improve accuracy but increase memory usage, complicating deployment in resource-constrained environments.

Feedback loops are inherent to this stage. Insights from deployment inform adjustments to models, while performance on test sets guides future data collection and annotation. Understanding these cycles is critical for iterative improvement and long-term success.

Scaling model development introduces emergent behaviors, such as bottlenecks in shared resources or unexpected interactions between multiple training experiments. Addressing these behaviors requires robust planning and the ability to anticipate system-wide patterns that might arise from local changes.

The boundaries between model development and other lifecycle stages often blur. Feature engineering overlaps with data preparation, while optimization for inference spans both development and deployment. Navigating these overlaps effectively requires careful coordination and clear interface definitions.

5.5.5 Lifecycle Implications

Model development is not an isolated task; it exists within the broader ML lifecycle. Decisions made here influence data preparation strategies, training infrastructure, and deployment feasibility. The iterative nature of this stage ensures that insights gained feed back into data collection and system optimization, reinforcing the interconnectedness of the lifecycle.

In subsequent chapters, we will explore key questions that arise during model development:

- How can scalable training infrastructures be designed for large-scale ML models?
- What frameworks and tools help manage the complexity of distributed training?
- How can model reproducibility and version control be ensured in evolving projects?
- What trade-offs must be made to balance accuracy with operational constraints?
- How can continual learning and updates be handled in production systems?

These questions highlight how model development sits at the core of ML systems, with decisions in this stage resonating throughout the entire lifecycle.

5.6 Deployment

Once validated, the trained model is integrated into production systems and workflows. Deployment requires addressing practical challenges such as system compatibility, scalability, and operational constraints. Successful integration hinges on ensuring that the model's predictions are not only accurate but also actionable in real-world settings, where resource limitations and workflow disruptions can pose significant barriers.

In the DR project, deployment strategies were shaped by the diverse environments in which the system would operate. Edge deployment enabled local processing of retinal images in rural clinics with intermittent connectivity, while automated quality checks flagged poor-quality images for recapture, ensuring reliable predictions. These measures demonstrate how deployment must bridge technological sophistication with usability and scalability across varied clinical settings.

5.6.1 Deployment Requirements and Impact

The requirements for deployment stem from both the technical specifications of the model and the operational constraints of its intended environment. In the DR project, the model needed to operate in rural clinics with limited computational resources and intermittent internet connectivity. Additionally, it had to fit seamlessly into the existing clinical workflow, which required rapid, interpretable results that could assist healthcare providers without causing disruption.

These requirements influenced deployment strategies significantly. A cloud-based deployment, while technically simpler, was not feasible due to unreliable connectivity in many clinics. Instead, the team opted for edge deployment, where models ran locally on clinic hardware. This approach required optimizing the model for smaller, less powerful devices while maintaining high accuracy. Optimization techniques such as model quantization and pruning were employed to reduce resource demands without sacrificing performance.

Integration with existing systems posed additional challenges. The ML system had to interface with hospital information systems (HIS) for accessing patient records and storing results. Privacy regulations mandated secure data handling at every step, further shaping deployment decisions. These considerations ensured that the system adhered to clinical and legal standards while remaining practical for daily use.

5.6.2 Deployment Workflow

The deployment and integration workflow in the DR project highlighted the interplay between model functionality, infrastructure, and user experience. The process began with thorough testing in simulated environments that replicated the technical constraints and workflows of the target clinics. These simulations helped identify potential bottlenecks and incompatibilities early, allowing the team to refine the deployment strategy before full-scale rollout.

Once the deployment strategy was finalized, the team implemented a phased rollout. Initial deployments were limited to a few pilot sites, allowing for controlled testing in real-world conditions. This approach provided valuable feedback from clinicians and technical staff, helping to identify issues that hadn't surfaced during simulations.

Integration efforts focused on ensuring seamless interaction between the ML system and existing tools. For example, the DR system had to pull patient information from the HIS, process retinal images from connected cameras, and return results in a format that clinicians could easily interpret. These tasks required the development of robust APIs, real-time data processing pipelines, and user-friendly interfaces tailored to the needs of healthcare providers.

5.6.3 Scale and Distribution

Scaling deployment across multiple locations introduced new complexities. Each clinic had unique infrastructure, ranging from differences in imaging equipment to variations in network reliability. These differences necessitated flexible deployment strategies that could adapt to diverse environments while ensuring consistent performance.

Despite achieving high performance metrics during development, the DR system faced unexpected challenges in real-world deployment. For example, in rural clinics, variations in imaging equipment and operator expertise led to inconsistencies in image quality that the model struggled to handle. These issues underscored the gap between laboratory success and operational reliability, prompting iterative refinements in both the model and the deployment strategy. Feedback from clinicians further revealed that initial system interfaces were not intuitive enough for widespread adoption, leading to additional redesigns.

Distribution challenges extended beyond infrastructure variability. The team needed to maintain synchronized updates across all deployment sites to ensure that improvements in model performance or system features were universally applied. This required implementing centralized version control systems and automated update pipelines that minimized disruption to clinical operations.

Despite achieving high performance metrics during development, the DR system faced unexpected challenges in real-world deployment. As illustrated in Figure 5.5, these challenges create multiple feedback paths—"Deployment Constraints" flowing back to model training to trigger optimizations, while "Performance Insights" from monitoring could necessitate new data collection. For example, when the system struggled with images from older camera models, this triggered both model optimizations and targeted data collection to improve performance under these conditions.

Another critical scaling challenge was training and supporting end-users. Clinicians and staff needed to understand how to operate the system, interpret its outputs, and provide feedback. The team developed comprehensive training programs and support channels to facilitate this transition, recognizing that user trust and proficiency were essential for system adoption.

5.6.4 Robustness and Reliability

In a clinical context, reliability is paramount. The DR system needed to function seamlessly under a wide range of conditions, from high patient volumes to suboptimal imaging setups. To ensure robustness, the team implemented fail-safes that could detect and handle common issues, such as incomplete or poor-quality data. These mechanisms included automated image quality checks and fallback workflows for cases where the system encountered errors.

Testing played a central role in ensuring reliability. The team conducted extensive stress testing to simulate peak usage scenarios, validating that the system could handle high throughput without degradation in performance. Redundancy was built into critical components to minimize the risk of downtime, and all interactions with external systems, such as the HIS, were rigorously tested for compatibility and security.

5.6.5 Systems Thinking

Deployment and integration, viewed through a systems lens, reveal deep connections to every other stage of the ML lifecycle. Decisions made during model development influence deployment architecture, while choices about data handling affect integration strategies. Monitoring requirements often dictate how deployment pipelines are structured, ensuring compatibility with real-time feedback loops.

Feedback loops are integral to deployment and integration. Real-world usage generates valuable insights that inform future iterations of model development and evaluation. For example, clinician feedback on system usability during the DR project highlighted the need for clearer interfaces and more interpretable outputs, prompting targeted refinements in design and functionality.

Emergent behaviors frequently arise during deployment. In the DR project, early adoption revealed unexpected patterns, such as clinicians using the system for edge cases or non-critical diagnostics. These behaviors, which were not predicted during development, necessitated adjustments to both the system's operational focus and its training programs.

Deployment introduces significant resource dependencies. Running ML models on edge devices required balancing computational efficiency with accuracy, while ensuring other clinic operations were not disrupted. These trade-offs extended to the broader system, influencing everything from hardware requirements to scheduling updates without affecting clinical workflows.

The boundaries between deployment and other lifecycle stages are fluid. Optimization efforts for edge devices often overlapped with model development, while training programs for clinicians fed directly into monitoring and maintenance. Navigating these overlaps required clear communication and collaboration between teams, ensuring seamless integration and ongoing system adaptability.

By applying a systems perspective to deployment and integration, we can better anticipate challenges, design robust solutions, and maintain the flexibility needed to adapt to evolving operational and technical demands. This approach ensures that ML systems not only achieve initial success but remain effective and reliable in real-world applications.

5.6.6 Lifecycle Implications

Deployment and integration are not terminal stages; they are the point at which an ML system becomes operationally active and starts generating real-world feedback. This feedback loops back into earlier stages, informing data collection strategies, model improvements, and evaluation protocols. By embedding lifecycle thinking into deployment, teams can design systems that are not only operationally effective but also adaptable and resilient to evolving needs.

In subsequent chapters, we will explore key questions related to deployment and integration:

- How can deployment strategies balance computational constraints with performance needs?
- What frameworks support scalable, synchronized deployments across diverse environments?
- How can systems be designed for seamless integration with existing workflows and tools?
- What are best practices for ensuring user trust and proficiency in operating ML systems?
- How do deployment insights feed back into the ML lifecycle to drive continuous improvement?

These questions emphasize the interconnected nature of deployment and integration within the lifecycle, highlighting the importance of aligning technical and operational priorities to create systems that deliver meaningful, lasting impact.

5.7 Maintenance

Monitoring and maintenance represent the ongoing, critical processes that ensure the continued effectiveness and reliability of deployed machine learning systems. Unlike traditional software, ML systems must account for shifts in data distributions, changing usage patterns, and evolving operational requirements. Monitoring provides the feedback necessary to adapt to these challenges, while maintenance ensures the system evolves to meet new needs.

As shown in Figure 5.5, monitoring serves as a central hub for system improvement, generating three critical feedback loops: "Performance Insights" flowing back to data collection to address gaps, "Data Quality Issues" triggering refinements in data preparation, and "Model Updates" initiating retraining when performance drifts. In the DR project, these feedback loops enabled continuous system improvement, from identifying underrepresented patient demographics (triggering new data collection) to detecting image quality issues (improving preprocessing) and addressing model drift (initiating retraining).

For DR screening, continuous monitoring tracked system performance across diverse clinics, detecting issues such as changing patient demographics or new imaging technologies that could impact accuracy. Proactive maintenance included plans to incorporate 3D imaging modalities like OCT, expanding the system's capabilities to diagnose a wider range of conditions. This highlights the importance of designing systems that can adapt to future challenges while maintaining compliance with rigorous healthcare regulations.

5.7.1 Monitoring Requirements and Impact

The requirements for monitoring and maintenance emerged from both technical needs and operational realities. In the DR project, the technical perspective required continuous tracking of model performance, data quality, and system resource usage. However, operational constraints added layers of complexity: monitoring systems had to align with clinical workflows, detect shifts in patient demographics, and provide actionable insights to both technical teams and healthcare providers.

Initial deployment highlighted several areas where the system failed to meet real-world needs, such as decreased accuracy in clinics with outdated equipment or lower-quality images. Monitoring systems detected performance drops in specific subgroups, such as patients with less common retinal conditions, demonstrating that even a well-trained model could face blind spots in practice. These insights informed maintenance strategies, including targeted updates to address specific challenges and expanded training datasets to cover edge cases.

These requirements influenced system design significantly. The critical nature of the DR system's function demanded real-time monitoring capabilities rather than periodic offline evaluations. To support this, the team implemented advanced logging and analytics pipelines to process large amounts of operational data from clinics without disrupting diagnostic workflows. Secure and efficient data handling was essential to transmit data across multiple clinics while preserving patient confidentiality.

Monitoring requirements also affected model design, as the team incorporated mechanisms for granular performance tracking and anomaly detection.

Even the system's user interface was influenced, needing to present monitoring data in a clear, actionable manner for clinical and technical staff alike.

5.7.2 Maintenance Workflow

The monitoring and maintenance workflow in the DR project revealed the intricate interplay between automated systems, human expertise, and evolving healthcare practices. The process began with defining a comprehensive monitoring framework, establishing key performance indicators (KPIs), and implementing dashboards and alert systems. This framework had to balance depth of monitoring with system performance and privacy considerations, collecting sufficient data to detect issues without overburdening the system or violating patient confidentiality.

As the system matured, maintenance became an increasingly dynamic process. Model updates driven by new medical knowledge or performance improvements required careful validation and controlled rollouts. The team employed A/B testing frameworks[7] to evaluate updates in real-world conditions and implemented rollback mechanisms to address issues quickly when they arose.

Monitoring and maintenance formed an iterative cycle rather than discrete phases. Insights from monitoring informed maintenance activities, while maintenance efforts often necessitated updates to monitoring strategies. The team developed workflows to transition seamlessly from issue detection to resolution, involving collaboration across technical and clinical domains.

5.7.3 Scale and Distribution

As the DR project scaled from pilot sites to widespread deployment, monitoring and maintenance complexities grew exponentially. Each additional clinic added to the volume of operational data and introduced new environmental variables, such as differing hardware configurations or demographic patterns.

The need to monitor both global performance metrics and site-specific behaviors required sophisticated infrastructure. While global metrics provided an overview of system health, localized issues, including a hardware malfunction at a specific clinic or unexpected patterns in patient data, needed targeted monitoring. Advanced analytics systems processed data from all clinics to identify these localized anomalies while maintaining a system-wide perspective.

Continuous adaptation added further complexity. Real-world usage exposed the system to an ever-expanding range of scenarios. Capturing insights from these scenarios and using them to drive system updates required efficient mechanisms for integrating new data into training pipelines and deploying improved models without disrupting clinical workflows.

5.7.4 Proactive Maintenance

Reactive maintenance alone was insufficient for the DR project's dynamic operating environment. Proactive strategies became essential to anticipate and prevent issues before they affected clinical operations.

[7] A/B Testing: A method in statistics to compare two versions of a variable to determine which performs better in a controlled environment.

The team implemented predictive maintenance models to identify potential problems based on patterns in operational data. Continuous learning pipelines allowed the system to retrain and adapt based on new data, ensuring its relevance as clinical practices or patient demographics evolved. These capabilities required careful balancing to ensure safety and reliability while maintaining system performance.

Metrics assessing adaptability and resilience became as important as accuracy, reflecting the system's ability to evolve alongside its operating environment. Proactive maintenance ensured the system could handle future challenges without sacrificing reliability.

5.7.5 Systems Thinking

Monitoring and maintenance, viewed through a systems lens, reveal their deep integration with every other stage of the ML lifecycle. Changes in data collection affect model behavior, which influences monitoring thresholds. Maintenance actions can alter system availability or performance, impacting users and clinical workflows.

Feedback loops are central to these processes. Monitoring insights drive updates to models and workflows, while user feedback informs maintenance priorities. These loops ensure the system remains responsive to both technical and clinical needs.

Emergent behaviors often arise in distributed deployments. The DR team identified subtle system-wide shifts in diagnostic patterns that were invisible in individual clinics but evident in aggregated data. Managing these behaviors required sophisticated analytics and a holistic view of the system.

Resource dependencies also presented challenges. Real-time monitoring competed with diagnostic functions for computational resources, while maintenance activities required skilled personnel and occasional downtime. Effective resource planning was critical to balancing these demands.

5.7.6 Lifecycle Implications

Monitoring and maintenance are not isolated stages but integral parts of the ML lifecycle. Insights gained from these activities feed back into data collection, model development, and evaluation, ensuring the system evolves in response to real-world challenges. This lifecycle perspective emphasizes the need for strategies that not only address immediate concerns but also support long-term adaptability and improvement.

In subsequent chapters, we will explore critical questions related to monitoring and maintenance:

- How can monitoring systems detect subtle degradations in ML performance across diverse environments?
- What strategies support efficient maintenance of ML systems deployed at scale?
- How can continuous learning pipelines ensure relevance without compromising safety?

- What tools facilitate proactive maintenance and minimize disruption in production systems?
- How do monitoring and maintenance processes influence the design of future ML models?

These questions highlight the interconnected nature of monitoring and maintenance, where success depends on creating a framework that ensures both immediate reliability and long-term viability in complex, dynamic environments.

5.8 AI Lifecycle Roles

Building effective and resilient machine learning systems is far more than a solo pursuit; it's a collaborative endeavor that thrives on the diverse expertise of a multidisciplinary team. Each role in this intricate dance brings unique skills and insights, supporting different phases of the AI development process. Understanding who these players are, what they contribute, and how they interconnect is crucial to navigating the complexities of modern AI systems.

5.8.1 Collaboration in AI

At the heart of any AI project is a team of data scientists. These innovative thinkers focus on model creation, experiment with architectures, and refine the algorithms that will become the neural networks driving insights from data. In our DR project, data scientists were instrumental in architecting neural networks capable of identifying retinal anomalies, advancing through iterations to fine-tune a balance between accuracy and computational efficiency.

Behind the scenes, data engineers work tirelessly to design robust data pipelines, ensuring that vast amounts of data are ingested, transformed, and stored effectively. They play a crucial role in the DR project, handling data from various clinics and automating quality checks to guarantee that the training inputs were standardized and reliable.

Meanwhile, machine learning engineers take the baton to integrate these models into production settings. They guarantee that models are nimble, scalable, and fit the constraints of the deployment environment. In rural clinics where computational resources can be scarce, their work in optimizing models was pivotal to enabling on-the-spot diagnosis.

Domain experts, such as ophthalmologists in the DR project, infuse technical progress with practical relevance. Their insights shape early problem definitions and ensure that AI tools align closely with real-world needs, offering a measure of validation that keeps the outcome aligned with clinical and operational realities.

MLOps engineers are the guardians of workflow automation, orchestrating the continuous integration and monitoring systems that keep AI models up and running. They crafted centralized monitoring frameworks in the DR project, ensuring that updates were streamlined and model performance remained optimal across different deployment sites.

Ethicists and compliance officers remind us of the larger responsibility that accompanies AI deployment, ensuring adherence to ethical standards and legal requirements. Their oversight in the DR initiative safeguarded patient privacy amidst strict healthcare regulations.

Project managers weave together these diverse strands, orchestrating timelines, resources, and communication streams to maintain project momentum and alignment with objectives. They acted as linchpins within the project, harmonizing efforts between tech teams, clinical practitioners, and policy makers.

5.8.2 Role Interplay

The synergy between these roles fuels the AI machinery toward successful outcomes. Data engineers establish a solid foundation for data scientists' creative model-building endeavors. As models transition into real-world applications, ML engineers ensure compatibility and efficiency. Meanwhile, feedback loops between MLOps engineers and data scientists foster continuous improvement, enabling quick adaptation to data-driven discoveries.

Ultimately, the success of the DR project underscores the irreplaceable value of interdisciplinary collaboration. From bridging clinical insights with technical prowess to ensuring ethical deployment, this collective effort exemplifies how AI initiatives can be both technically successful and socially impactful.

This interconnected approach underlines why our exploration in later chapters will delve into various aspects of AI development, including those that may be seen as outside an individual's primary expertise. Understanding these diverse roles will equip us to build more robust, well-rounded AI solutions. By comprehending the broader context and the interplay of roles, you'll be better prepared to address challenges and collaborate effectively, paving the way for innovative and responsible AI systems.

5.9 Conclusion

The AI workflow we've explored, while illustrated through the Diabetic Retinopathy project, represents a framework applicable across diverse domains of AI application. From finance and manufacturing to environmental monitoring and autonomous vehicles, the core stages of the workflow remain consistent, even as their specific implementations vary widely.

The interconnected nature of the AI lifecycle, illustrated in Figure 5.5, is a universal constant. The feedback loops, from "Performance Insights" driving data collection to "Validation Issues" triggering model updates, demonstrate how decisions in one stage invariably impact others. Data quality affects model performance, deployment constraints influence architecture choices, and real-world usage patterns drive ongoing refinement through these well-defined feedback paths.

Regardless of the application, the interconnected nature of the AI lifecycle is a universal constant. Whether developing fraud detection systems for banks or predictive maintenance models for industrial equipment, decisions made in one stage invariably impact others. Data quality affects model performance, deployment constraints influence architecture choices, and real-world usage patterns drive ongoing refinement.

This interconnectedness underscores the importance of systems thinking in AI development across all sectors. Success in AI projects, regardless of domain, comes from understanding and managing the complex interactions between stages, always considering the broader context in which the system will operate.

As AI continues to evolve and expand into new areas, this holistic approach becomes increasingly crucial. Future challenges in AI development, whether in healthcare, finance, environmental science, or any other field, will likely center around managing increased complexity, ensuring adaptability, and balancing performance with ethical considerations. By approaching AI development with a systems-oriented mindset, we can create solutions that are not only technically proficient but also robust, adaptable, and aligned with real-world needs across a wide spectrum of applications.

5.10 Resources

ℹ Slides

- *Coming soon.*

! Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 6

Data Engineering

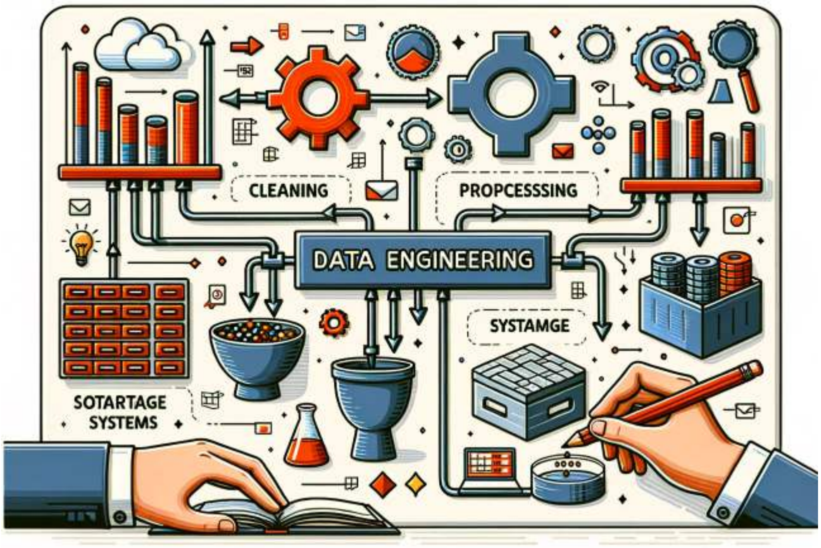


Figure 6.1: *DALL·E 3 Prompt: Create a rectangular illustration visualizing the concept of data engineering. Include elements such as raw data sources, data processing pipelines, storage systems, and refined datasets. Show how raw data is transformed through cleaning, processing, and storage to become valuable information that can be analyzed and used for decision-making.*

Purpose

How does data shape ML systems engineering?

In the field of machine learning, data engineering is often overshadowed by the allure of sophisticated algorithms, when in fact data plays a foundational role in determining an AI system's capabilities and limitations. We need to understand the core principles of data in ML systems, exploring how the acquisition, processing, storage, and governance of data directly impact the performance, reliability, and ethical considerations of AI systems. By understanding these fundamental concepts, we can unlock the true potential of AI and build a solid foundation of high-quality ML solutions.

Learning Objectives

- Analyze different data sourcing methods (datasets, web scraping, crowdsourcing, synthetic data).
- Explain the importance of data labeling and ensure label quality.
- Evaluate data storage systems for ML workloads (databases, data warehouses, data lakes).
- Describe the role of data pipelines in ML systems.
- Explain the importance of data governance in ML (security, privacy, ethics).
- Identify key challenges in data engineering for ML.

6.1 Overview

Data is the foundation of modern machine learning systems, as success is governed by the quality and accessibility of training and evaluation data. Despite its pivotal role, data engineering is often overlooked compared to algorithm design and model development. However, the effectiveness of any machine learning system hinges on the robustness of its data pipeline. As machine learning applications become more sophisticated, the challenges associated with curating, cleaning, organizing, and storing data have grown significantly. These activities have emerged as some of the most resource-intensive aspects of the data engineering process, requiring sustained effort and attention.

Definition of Data Engineering

Data Engineering is the *process of designing, building, and maintaining* the infrastructure and systems that *collect, store, and process* data for analysis and machine learning. It involves *data acquisition, transformation, and management*, ensuring data is *reliable, accessible, and optimized* for downstream applications. Data engineering focuses on *building robust data pipelines* and architectures that support the *efficient and scalable handling* of large datasets.

The concept of “Data Cascades,” introduced by Sambasivan et al. (2021), highlights the systemic failures that can arise when data quality issues are left unaddressed. Errors originating during data collection or processing stages can compound over time, creating cascading effects that lead to model failures, costly retraining, or even project termination. The failures of IBM Watson Health in 2019, where flawed training data resulted in unsafe and incorrect cancer treatment recommendations (Strickland 2019), show the real-world consequences of neglecting data quality and its associated engineering requirements.

It is therefore unsurprising that data scientists spend the majority of their time, up to 60% as shown in Figure 6.2, is spent on cleaning and organizing data. This statistic highlights the critical need to prioritize data-related challenges early in the pipeline to avoid downstream issues and ensure the effectiveness of machine learning systems.

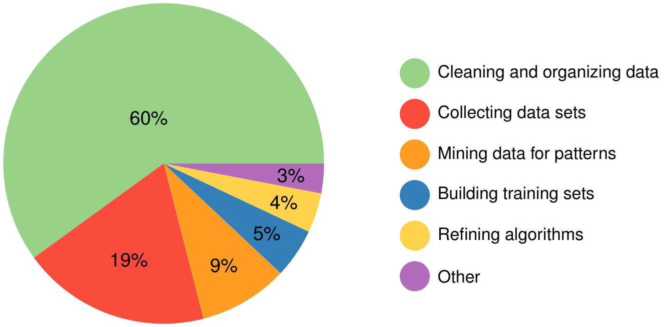


Figure 6.2: Data scientists spend most of their time on cleaning and organizing data.

Data engineering encompasses multiple critical stages in machine learning systems, from initial data collection through processing and storage. The discussion begins with the identification and sourcing of data, exploring diverse origins such as pre-existing datasets, web scraping, crowdsourcing, and synthetic data generation. Special attention is given to the complexities of integrating heterogeneous sources, validating incoming data, and handling errors during ingestion.

Next, the exploration covers the transformation of raw data into machine learning-ready formats. This process involves cleaning, normalizing, and extracting features, tasks that are critical to optimizing model learning and ensuring robust performance. The challenges of scale and computational efficiency are also discussed, as they are particularly important for systems that operate on vast and complex datasets.

Beyond data processing, the text addresses the intricacies of data labeling, a crucial step for supervised learning systems. Effective labeling requires sound annotation methodologies and advanced techniques such as AI-assisted annotation to ensure the accuracy and consistency of labeled data. Challenges such as bias and ambiguity in labeling are explored, with examples illustrating their potential impact on downstream tasks.

The discussion also examines the storage and organization of data, a vital aspect of supporting machine learning pipelines across their lifecycle. Topics such as storage system design, feature stores, caching strategies, and access patterns are discussed, with a focus on ensuring scalability and efficiency. Governance is highlighted as a key component of data storage and management, emphasizing the importance of compliance with privacy regulations, ethical considerations, and the use of documentation frameworks to maintain transparency and accountability.

This chapter provides an exploration of data engineering practices necessary for building and maintaining effective machine learning systems. The end goal is to emphasize the often-overlooked importance of data in enabling the success of machine learning applications.

6.2 Problem Definition

As discussed in the overview, Sambasivan et al. (2021) observes that neglecting the fundamental importance of data quality gives rise to "Data Cascades" — events where lapses in data quality compound, leading to negative downstream consequences such as flawed predictions, project terminations, and even potential harm to communities. Despite many ML professionals recognizing the importance of data, numerous practitioners report facing these cascades.

Figure 6.3 illustrates these potential data pitfalls at every stage and how they influence the entire process down the line. The influence of data collection errors is especially pronounced. As illustrated in the figure, any lapses in this initial stage will become apparent at later stages (in model evaluation and deployment) and might lead to costly consequences, such as abandoning the entire model and restarting anew. Therefore, investing in data engineering techniques from the onset will help us detect errors early, mitigating these cascading effects.

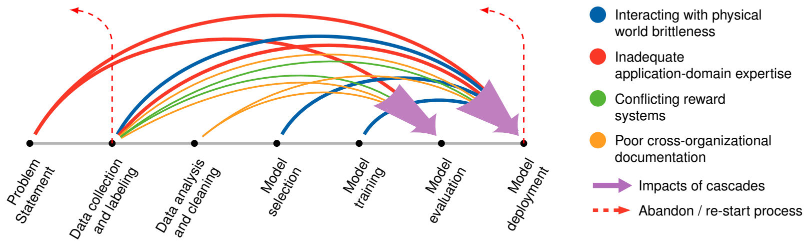

Problem Statement
Data collection and labeling
Data analysis and cleaning
Model selection
Model training
Model evaluation
Model deployment
Interacting with physical world brittleness
Inadequate application-domain expertise
Conflicting reward systems
Poor cross-organizational documentation
Impacts of cascades
Abandon / re-start process

Figure 6.3: Data cascades: compounded costs. Source: Sambasivan et al. (2021).

This emphasis on data quality and proper problem definition is fundamental across all types of ML systems. As Sculley et al. (2015) emphasize, it is important to distinguish ML-specific problem framing from the broader context of general software development. Whether developing recommendation engines processing millions of user interactions, computer vision systems analyzing medical images, or natural language models handling diverse text data, each system brings unique challenges that must be carefully considered from the outset. Production ML systems are particularly sensitive to data quality issues, as they must handle continuous data streams, maintain consistent processing pipelines, and adapt to evolving patterns while maintaining performance standards.

A solid project foundation is essential for setting the trajectory and ensuring the eventual success of any initiative. At the heart of this foundation lies the crucial first step: identifying a clear problem to solve. This could involve challenges like developing a recommendation system that effectively handles cold-start scenarios, or creating a classification model that maintains consistent accuracy across diverse population segments.

As we will explore later in this chapter, establishing clear objectives provides a unified direction that guides the entire project. These objectives might include creating representative datasets that account for various real-world scenarios. Equally important is defining specific benchmarks, such as prediction accuracy and system latency, which offer measurable outcomes to gauge progress and success.

Throughout this process, engaging with stakeholders, including end-users and business leaders, provides invaluable insights that ensure the project remains aligned with real-world needs and expectations.

In particular, a cardinal sin in ML is to begin collecting data (or augmenting an existing dataset) without clearly specifying the underlying problem definition to guide the data collection. We identify the key steps that should precede any data collection effort here:

1. Identify and clearly state the problem definition
2. Set clear objectives to meet
3. Establish success benchmarks
4. Understand end-user engagement/use
5. Understand the constraints and limitations of deployment
6. Perform data collection.
7. Iterate and refine.

6.2.1 Keyword Spotting Example

Keyword Spotting (KWS) is an excellent example to illustrate all of the general steps in action. This technology is critical for voice-enabled interfaces on endpoint devices such as smartphones. Typically functioning as lightweight wake-word engines, KWS systems are constantly active, listening for a specific phrase to trigger further actions.

As shown in Figure 6.4, when we say "OK, Google" or "Alexa," this initiates a process on a microcontroller embedded within the device.

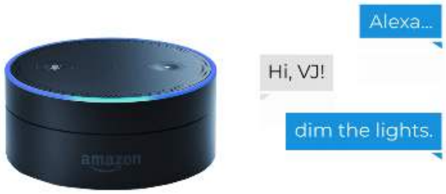


Figure 6.4: Keyword Spotting example: interacting with Alexa. Source: Amazon.

Building a reliable KWS model is a complex task. It demands a deep understanding of the deployment scenario, encompassing where and how these devices will operate. For instance, a KWS model's effectiveness is not just about recognizing a word; it's about discerning it among various accents and background noises, whether in a bustling cafe or amid the blaring sound of a television in a living room or a kitchen where these devices are commonly found. It's about ensuring that a whispered "Alexa" in the dead of night or a shouted "OK Google" in a noisy marketplace are recognized with equal precision.

Moreover, many current KWS voice assistants support a limited number of languages, leaving a substantial portion of the world's linguistic diversity unrepresented. This limitation is partly due to the difficulty in gathering and monetizing data for languages spoken by smaller populations. In the long-tail distribution of languages, most languages have limited or zero speech training data available, making the development of voice assistants challenging.

Keyword spotting models can run on low-power, low-price microcontrollers, so theoretically voice interfaces could be expanded to a huge gamut of devices worldwide, beyond smartphones and home assistants. But the level of accuracy and robustness that end-users expect hinges on the availability and quality of speech data, and the ability to label the data correctly. Developing a keyword-spotting model for an arbitrary word or phrase in an arbitrary language begins with clearly understanding the problem statement or definition. Using KWS as an example, we can break down each of the steps as follows:

1. **Identifying the Problem**: KWS detects specific keywords amidst ambient sounds and other spoken words. The primary problem is to design a system that can recognize these keywords with high accuracy, low latency, and minimal false positives or negatives, especially when deployed on devices with limited computational resources. A well-specified problem definition for developing a new KWS model should identify the desired keywords along with the envisioned application and deployment scenario.
2. **Setting Clear Objectives**: The objectives for a KWS system might include:
   - Achieving a specific accuracy rate (e.g., 98% accuracy in keyword detection).
   - Ensuring low latency (e.g., keyword detection and response within 200 milliseconds).
   - Minimizing power consumption to extend battery life on embedded devices.
   - Ensuring the model's size is optimized for the available memory on the device.
3. **Benchmarks for Success**: Establish clear metrics to measure the success of the KWS system. This could include:
   - *True Positive Rate:* The percentage of correctly identified keywords relative to all spoken keywords.
   - *False Positive Rate:* The percentage of non-keywords (including silence, background noise, and out-of-vocabulary words) incorrectly identified as keywords.
   - *Detection/Error Tradeoff* These curves evaluate KWS on streaming audio representative of a real-world deployment scenario, by comparing the number of false accepts per hour (the number of false positives over the total duration of the evaluation audio) against the false rejection rate (the number of missed keywords relative to the number of spoken keywords in the evaluation audio). Nayak et al. (2022) provides one example of this.
   - *Response Time:* The time taken from keyword utterance to system response.
   - *Power Consumption:* Average power used during keyword detection.

4. **Stakeholder Engagement and Understanding**: Engage with stakeholders, which include device manufacturers, hardware and software developers, and end-users. Understand their needs, capabilities, and constraints. For instance:
   - Device manufacturers might prioritize low power consumption.
   - Software developers might emphasize ease of integration.
   - End-users would prioritize accuracy and responsiveness.
5. **Understanding the Constraints and Limitations of Embedded Systems**: Embedded devices come with their own set of challenges:
   - *Memory Limitations:* KWS models must be lightweight to fit within the memory constraints of embedded devices. Typically, KWS models need to be as small as 16 KB to fit in the always-on island[0] of the SoC. Moreover, this is just the model size. Additional application code for preprocessing may also need to fit within the memory constraints.
   - Processing Power: The computational capabilities of embedded devices are limited (a few hundred MHz of clock speed), so the KWS model must be optimized for efficiency.
   - *Power Consumption:* Since many embedded devices are battery-powered, the KWS system must be power-efficient.
   - *Environmental Challenges:* Devices might be deployed in various environments, from quiet bedrooms to noisy industrial settings. The KWS system must be robust enough to function effectively across these scenarios.
6. **Data Collection and Analysis**: For a KWS system, the quality and diversity of data are paramount. Considerations might include:
   - *Demographics:* Collect data from speakers with various accents across age and gender to ensure wide-ranging recognition support.
   - *Keyword Variations:* People might pronounce keywords differently or express slight variations in the wake word itself. Ensure the dataset captures these nuances.
   - *Background Noises:* Include or augment data samples with different ambient noises to train the model for real-world scenarios.
7. **Iterative Feedback and Refinement**: Once a prototype KWS system is developed, it is important to do the following to ensure that the system remains aligned with the defined problem and objectives as the deployment scenarios change over time and as use-cases evolve.
   - Test it in real-world scenarios
   - Gather feedback - are some users or deployment scenarios encountering underperformance relative to others?
   - Iteratively refine the dataset and model

The KWS example illustrates the broader principles of problem definition, showing how initial decisions about data requirements ripple throughout a

[0] Always-on Island: A specialized, low-power subsystem within an SoC that continuously monitors sensors and manages wake-up functions. It enables efficient power management by keeping only essential components active while allowing rapid system wake-up when needed.

project's lifecycle. By carefully considering each aspect, from core problem identification, through performance benchmarks, to deployment constraints, teams can build a strong foundation for their ML systems. The methodical problem definition process provides a framework applicable across the ML spectrum. Whether developing computer vision systems for medical diagnostics, recommendation engines processing millions of user interactions, or natural language models analyzing diverse text corpora, this structured approach helps teams anticipate and plan for their data needs.

This brings us to data pipelines, the foundational infrastructure that transforms raw data into ML-ready formats, while maintaining quality and reliability throughout the process. These pipelines implement our carefully defined requirements in production systems, handling everything from initial data ingestion to final feature generation.

6.3 Pipeline Basics

Modern machine learning systems depend on data pipelines to process massive amounts of data efficiently and reliably. For instance, recommendation systems at companies like Netflix process billions of user interactions daily, while autonomous vehicle systems must handle terabytes of sensor data in real-time. These pipelines serve as the backbone of ML systems, acting as the infrastructure through which raw data transforms into ML-ready training data.

These data pipelines are not simple linear paths but rather complex systems. They must manage data acquisition, transformation, storage, and delivery while ensuring data quality and system reliability. The design of these pipelines fundamentally shapes what is possible with an ML system.

ML data pipelines consist of several distinct layers: data sources, ingestion, processing, labeling, storage, and eventually ML training (Figure 6.5). Each layer plays a specific role in the data preparation workflow. The interactions between these layers are crucial to the system's overall effectiveness. The flow from raw data sources to ML training demonstrates the importance of maintaining data quality and meeting system requirements throughout the pipeline.

6.4 Data Sources

The first stage of the pipeline architecture sourcing appropriate data to meet the training needs. The quality and diversity of this data will fundamentally determine our ML system's learning and prediction capabilities and limitations. ML systems can obtain their training data through several different approaches, each with their own advantages and challenges. Let's examine each of these approaches in detail.

6.4.1 Existing Datasets

Platforms like Kaggle and UCI Machine Learning Repository provide ML practitioners with ready-to-use datasets that can jumpstart system development. These pre-existing datasets are particularly valuable when building ML systems as they offer immediate access to cleaned, formatted data with established

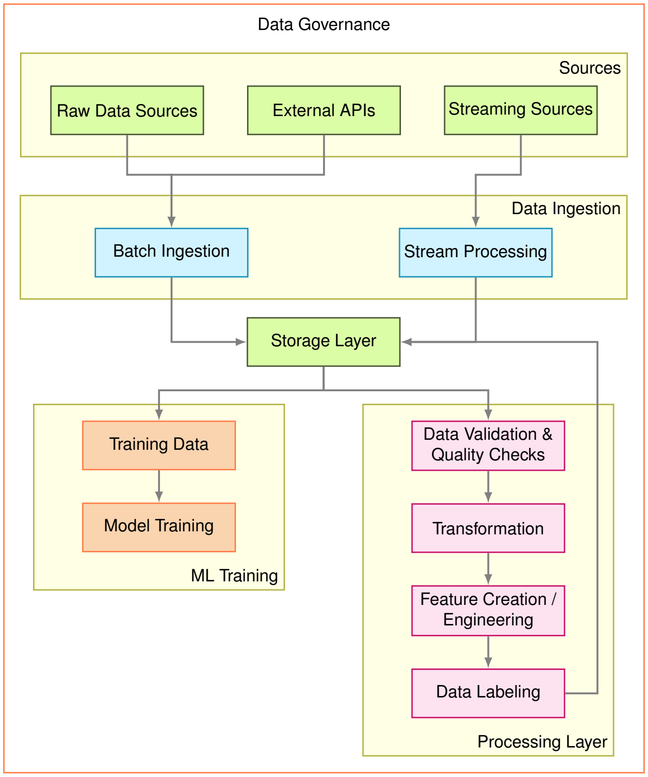


Figure 6.5: Overview of the data pipeline.

benchmarks. One of their primary advantages is cost efficiency, as creating datasets from scratch requires significant time and resources, especially when building production ML systems that need large amounts of high-quality training data.

Many of these datasets, such as ImageNet,[1] have become standard benchmarks in the machine learning community, enabling consistent performance comparisons across different models and architectures. For ML system developers, this standardization provides clear metrics for evaluating model improvements and system performance. The immediate availability of these datasets allows teams to begin experimentation and prototyping without delays in data collection and preprocessing.

However, ML practitioners must carefully consider the quality assurance aspects of pre-existing datasets. For instance, the ImageNet dataset was found to have label errors on 6.4% of the validation set (Northcutt, Athalye, and Mueller 2021). While popular datasets benefit from community scrutiny that helps identify and correct errors and biases, most datasets remain “untended gardens” where quality issues can significantly impact downstream system performance if not properly addressed. Moreover, as (Gebru et al. 2021a) highlighted in her paper, simply providing a dataset without documentation

[1] It is essential to be aware of the several limitations of these benchmark datasets that might not be immediately clear. For example, models trained on these datasets may be overly optimized to specifics in the dataset and will begin to overfit to these characteristics. Many benchmark datasets are not updated overtime which may make them outdated and biased towards a different time period.

can lead to misuse and misinterpretation, potentially amplifying biases present in the data.[2]

Supporting documentation accompanying existing datasets is invaluable, yet is often only present in widely-used datasets. Good documentation provides insights into the data collection process and variable definitions and sometimes even offers baseline model performances. This information not only aids understanding but also promotes reproducibility in research, a cornerstone of scientific integrity; currently, there is a crisis around improving reproducibility in machine learning systems (Pineau et al. 2021). When other researchers have access to the same data, they can validate findings, test new hypotheses, or apply different methodologies, thus allowing us to build on each other's work more rapidly.

While existing datasets are invaluable resources, it's essential to understand the context in which the data was collected. Researchers should be wary of potential overfitting when using popular datasets such as ImageNet (Beyer et al. 2020), leading to inflated performance metrics. Sometimes, these datasets do not reflect the real-world data.

A key consideration for ML systems is how well pre-existing datasets reflect real-world deployment conditions. Relying on standard datasets can create a concerning disconnect between training and production environments. This misalignment becomes particularly problematic when multiple ML systems are trained on the same datasets (Figure 6.6), potentially propagating biases and limitations throughout an entire ecosystem of deployed models.

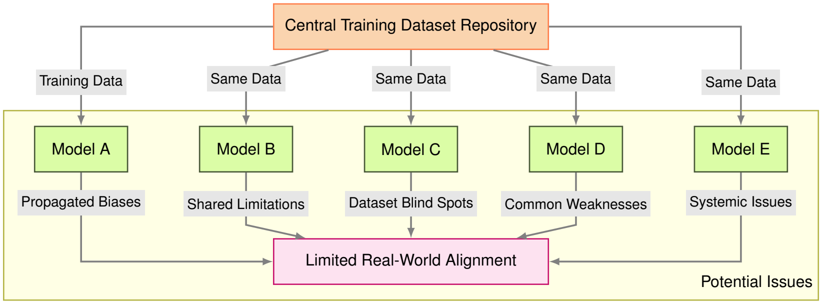


Figure 6.6: Training different models on the same dataset.

6.4.2 Web Scraping

When building ML systems, particularly in domains where pre-existing datasets are insufficient, web scraping offers a powerful approach to gathering training data at scale. This automated technique for extracting data from websites has become a powerful tool in modern ML system development. It enables teams to build custom datasets tailored to their specific needs.

Web scraping has proven particularly valuable for building large-scale ML systems when human-labeled data is scarce. Consider computer vision systems: major datasets like ImageNet and OpenImages were built through systematic web scraping, fundamentally advancing the field of computer vision. In production environments, companies regularly scrape e-commerce sites to gather

[2] Biases can infiltrate all stages of the ML workflow from data collection and feature engineering through model training and deployment. Each stage presents opportunities for bias to be introduced or amplified. The impacts of bias and approaches for mitigation are covered in depth in later chapters.

product images for recognition systems or social media platforms for computer vision applications. Stanford's LabelMe project demonstrated this approach's potential early on, scraping Flickr to create a diverse dataset of over 63,000 annotated images.

The impact of web scraping extends well beyond computer vision systems. In natural language processing, web-scraped data has enabled the development of increasingly sophisticated ML systems. Large language models, such as ChatGPT and Claude, rely on vast amounts of text scraped from the public internet and media to learn language patterns and generate responses (Groeneveld et al. 2024). Similarly, specialized ML systems like GitHub's Copilot demonstrate how targeted web scraping, in this case of code repositories, can create powerful domain-specific assistants (M. Chen et al. 2021).

Production ML systems often require continuous data collection to maintain relevance and performance. Web scraping facilitates this by gathering structured data like stock prices, weather patterns, or product information for analytical applications. However, this continuous collection introduces unique challenges for ML systems. Data consistency becomes crucial, as variations in website structure or content formatting can disrupt the data pipeline and affect model performance. Proper data management through databases or warehouses becomes essential not just for storage, but for maintaining data quality and enabling model updates.

Despite its utility, web scraping presents several challenges that ML system developers must carefully consider. Legal and ethical constraints can limit data collection, as not all websites permit scraping, and violating these restrictions can have serious consequences. When building ML systems with scraped data, teams must carefully document data sources and ensure compliance with terms of service and copyright laws. Privacy considerations become particularly critical when dealing with user-generated content, often requiring robust anonymization procedures.

Technical limitations also affect the reliability of web-scraped training data. Rate limiting by websites can slow data collection, while the dynamic nature of web content can introduce inconsistencies that impact model training. As shown in Figure 6.7, web scraping can yield unexpected or irrelevant data, for example, historical images appearing in contemporary image searches, that can pollute training datasets and degrade model performance. These issues highlight the importance of thorough data validation and cleaning processes in ML pipelines built on web-scraped data.

6.4.3 Crowdsourcing

Crowdsourcing is a collaborative approach to data collection, leveraging the collective efforts of distributed individuals via the internet to tackle tasks requiring human judgment[3]. By engaging a global pool of contributors, this method accelerates the creation of high-quality, labeled datasets for machine learning systems, especially in scenarios where pre-existing data is scarce or domain-specific. Platforms like Amazon Mechanical Turk exemplify how crowdsourcing facilitates this process by distributing annotation tasks to a global workforce. This enables the rapid collection of labels for complex tasks such as sentiment

[3] Crowdsourcing became popular for data labelling because it is cost effective and offers wide coverage across different demographics and cultures. However, this cost advantage often comes at the expense of the inadequate worker compensation, unpaid training time, and unstable income. Organizations using crowdsourcing platforms should prioritize ethical considerations related to the financial and emotional wellbeing of the workers.

Figure 6.7: A picture of old traffic lights (1914). Source: Vox.

analysis, image recognition, and speech transcription, significantly expediting the data preparation phase.

One of the most impactful examples of crowdsourcing in machine learning is the creation of the ImageNet dataset. ImageNet, which revolutionized computer vision, was built by distributing image labeling tasks to contributors via Amazon Mechanical Turk. The contributors categorized millions of images into thousands of classes, enabling researchers to train and benchmark models for a wide variety of visual recognition tasks.

The dataset's availability spurred advancements in deep learning, including the breakthrough AlexNet model in 2012, which demonstrated how large-scale, crowdsourced datasets could drive innovation. ImageNet's success highlights how leveraging a diverse group of contributors for annotation can enable machine learning systems to achieve unprecedented performance.

Another example of crowdsourcing's potential is Google's Crowdsource, a platform where volunteers contribute labeled data to improve AI systems in applications like language translation, handwriting recognition, and image understanding. By gamifying the process and engaging global participants, Google harnesses diverse datasets, particularly for underrepresented languages. This approach not only enhances the quality of AI systems but also empowers communities by enabling their contributions to influence technological development.

Crowdsourcing has also been instrumental in applications beyond traditional dataset annotation. For instance, the navigation app Waze uses crowdsourced data from its users to provide real-time traffic updates, route suggestions, and incident reporting. While this involves dynamic data collection rather than static dataset labeling, it demonstrates how crowdsourcing can generate continuously updated datasets essential for applications like mobile or edge ML systems. These systems often require real-time input to maintain relevance and accuracy in changing environments.

One of the primary advantages of crowdsourcing is its scalability. By distributing microtasks to a large audience, projects can process enormous volumes of data quickly and cost-effectively. This scalability is particularly beneficial for machine learning systems that require extensive datasets to achieve high performance. Additionally, the diversity of contributors introduces a wide range of perspectives, cultural insights, and linguistic variations, enriching datasets and improving models' ability to generalize across populations.

Flexibility is a key benefit of crowdsourcing. Tasks can be adjusted dynamically based on initial results, allowing for iterative improvements in data collection. For example, Google's reCAPTCHA system uses crowdsourcing to verify human users while simultaneously labeling datasets for training machine learning models. Users identify objects in images, including street signs and cars, contributing to the training of autonomous systems. This clever integration demonstrates how crowdsourcing can scale seamlessly when embedded into everyday workflows.

Despite its advantages, crowdsourcing presents challenges that require careful management. Quality control is a major concern, as the variability in contributors' expertise and attention can lead to inconsistent or inaccurate annotations. Providing clear instructions and training materials helps ensure participants understand the task requirements. Techniques such as embedding known test cases, leveraging consensus algorithms, or using redundant annotations can mitigate quality issues and align the process with the problem definition discussed earlier.

Ethical considerations are paramount in crowdsourcing, especially when datasets are built at scale using global contributors. It is essential to ensure that participants are fairly compensated for their work and that they are informed about how their contributions will be used. Additionally, privacy concerns must be addressed, particularly when dealing with sensitive or personal information. Transparent sourcing practices, clear communication with contributors, and robust auditing mechanisms are crucial for building trust and maintaining ethical standards.

The issue of fair compensation and ethical data sourcing was brought into sharp focus during the development of large-scale AI systems like OpenAI's ChatGPT. Reports revealed that OpenAI outsourced data annotation tasks to workers in Kenya, employing them to moderate content and identify harmful or inappropriate material that the model might generate. This involved reviewing and labeling distressing content, such as graphic violence and explicit material, to train the AI in recognizing and avoiding such outputs. While this approach enabled OpenAI to improve the safety and utility of ChatGPT, significant ethical concerns arose around the working conditions, the nature of the tasks, and the compensation provided to Kenyan workers.

Many of the contributors were reportedly paid as little as $1.32 per hour for reviewing and labeling highly traumatic material. The emotional toll of such work, coupled with low wages, raised serious questions about the fairness and transparency of the crowdsourcing process. This controversy highlights a critical gap in ethical crowdsourcing practices. The workers, often from economically disadvantaged regions, were not adequately supported to cope with the psychological impact of their tasks. The lack of mental health resources and insufficient compensation underscored the power imbalances that can emerge when outsourcing data annotation tasks to lower-income regions.

The challenges highlighted by the ChatGPT, particularly the Kenya controversy, are not unique to OpenAI. Many organizations that rely on crowdsourcing for data annotation face similar issues. As machine learning systems grow more complex and require larger datasets, the demand for annotated data will continue to increase. This shows the need for industry-wide standards and best practices to ensure ethical data sourcing. This case emphasizes the importance of considering the human labor behind AI systems. While crowdsourcing offers scalability and diversity, it also brings ethical responsibilities that cannot be overlooked. Organizations must prioritize the well-being and fair treatment of contributors as they build the datasets that drive AI innovation.

Moreover, when dealing with specialized applications like mobile ML, edge ML, or cloud ML, additional challenges may arise. These applications often require data collected from specific environments or devices, which can be difficult to gather through general crowdsourcing platforms. For example, data for mobile applications utilizing smartphone sensors may necessitate participants with specific hardware features or software versions. Similarly, edge ML systems deployed in industrial settings may require data involving proprietary processes or secure environments, introducing privacy and accessibility challenges.

Hybrid approaches that combine crowdsourcing with other data collection methods can address these challenges. Organizations may engage specialized communities, partner with relevant stakeholders, or create targeted initiatives to collect domain-specific data. Additionally, synthetic data generation, as discussed in the next section, can augment real-world data when crowdsourcing falls short.

6.4.4 Anonymization Techniques

Protecting the privacy of individuals while still enabling data-driven insights is a central challenge in the modern data landscape. As organizations collect and analyze vast quantities of information, the risk of exposing sensitive details, whether inadvertently or via intentional breaches, heightens. To mitigate these risks, practitioners have developed a commonly used range of anonymization techniques. These methods transform datasets such that individual identities and sensitive attributes become difficult or nearly impossible to re-identify, while preserving, to varying extents, the overall utility of the data for analysis.

Masking involves altering or obfuscating sensitive values so that they cannot be directly traced back to the original data subject. For instance, digits in financial account numbers or credit card numbers can be replaced with asterisks, a fixed set of dummy characters, or hashed values to protect sensitive information during display or logging. This anonymization technique is straightforward to implement and understand while clearly protecting identifiable values from being viewed, but may struggle with protecting broader context (e.g. relationships between data points).

Generalization reduces the precision or granularity of data to decrease the likelihood of re-identification. Instead of revealing an exact date of birth or address, the data is aggregated into broader categories (e.g., age ranges, zip code prefixes). For example, a user's exact age of 37 might be generalized to an age range of 30-39, while their exact address might be bucketed into a city level granularity. This technique clearly reduces the risk of identifying an individual by sharing data in aggregated form; however, we might consequently lose analytical prediction. Furthermore, if granularity is not chosen correctly, individuals may still be able to be identified under certain conditions.

Pseudonymization is the process of replacing direct identifiers (like names, Social Security numbers, or email addresses) with artificial identifiers, or "pseudonyms." These pseudonyms must not reveal, or be easily traceable to, the original data subject. This is commonly used in health records or in any situation where datasets need personal identities removed, but maintain unique entries. This approach allow maintaining individual-level data for analysis (since records can be traced through pseudonyms), while reducing the risk of direct identification. However, if the "key" linking the pseudonym to the real identifier is compromised, re-identification becomes possible.

k-anonymity ensures that each record in a dataset is indistinguishable from at least $k-1$ other records. This is achieved by suppressing or generalizing quasi-identifiers, or attributes that, in combination, could be used to re-identify an individual (e.g., zip code, age, gender). For example, if $k=5$, every record in the dataset must share the same combination of quasi-identifiers with at least four other records. Thus, an attacker cannot pinpoint a single individual simply by looking at these attributes. This approach provides a formal privacy guarantee that helps reduce chances of individual re-identification. However, it is extremely high touch and may require a significant level of data distortion and does not protect against things like homogeneity or background knowledge attacks.

Differential privacy (DP) adds carefully calibrated "noise" or randomized data perturbations to query results or datasets. The goal is to ensure that the inclusion or exclusion of any single individual's data does not significantly affect the output, thereby concealing their presence. Introduced noise is controlled by the ϵ parameter in ϵ-Differential Privacy, balancing data utility and privacy guarantees. The clear advantages this approach provides are strong mathematical guarantees of privacy, and DP is widely used in academic and industrial settings (e.g., large-scale data analysis). However, the added noise can affect data accuracy and subsequent model performance; proper parameter tuning is crucial to ensure both privacy and usefulness.

In summary, effective data anonymization is a balancing act between privacy and utility. Techniques such as masking, generalization, pseudonymization, k-anonymity, and differential privacy each target different aspects of re-identification risk. By carefully selecting and combining these methods, organizations can responsibly derive value from sensitive datasets while respecting the privacy rights and expectations of the individuals represented within them.

6.4.5 Synthetic Data Creation

Synthetic data generation has emerged as a powerful tool for addressing limitations in data collection, particularly in machine learning applications where real-world data is scarce, expensive, or ethically challenging to obtain. This approach involves creating artificial data using algorithms, simulations, or generative models to mimic real-world datasets. The generated data can be used to supplement or replace real-world data, expanding the possibilities for training robust and accurate machine learning systems. Figure 6.8 illustrates the process of combining synthetic data with historical datasets to create larger, more diverse training sets.

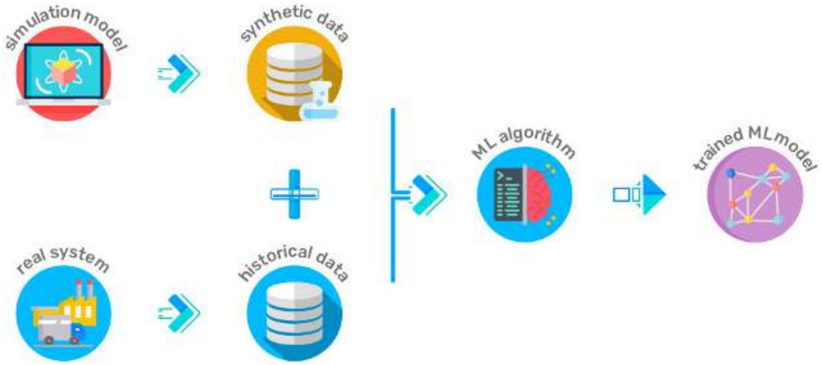


Figure 6.8: Increasing training data size with synthetic data generation. Source: AnyLogic.

Advancements in generative modeling techniques, such as diffusion models and flow-matching algorithms[4], Generative Adversarial Networks (GANs)[5], and Variational Autoencoders (VAEs)[6], have greatly enhanced the quality of synthetic data. These techniques can produce data that closely resembles real-world distributions, making it suitable for applications ranging from computer vision to natural language processing. For example, GANs have been used to generate synthetic images for object recognition tasks, creating diverse datasets that are almost indistinguishable from real-world images. Similarly, synthetic data has been leveraged to simulate speech patterns, enhancing the robustness of voice recognition systems.

Synthetic data has become particularly valuable in domains where obtaining real-world data is either impractical or costly. The automotive industry has embraced synthetic data to train autonomous vehicle systems; there are only so many cars you can physically crash to get crash-test data that might help an ML system know how to avoid crashes in the first place. Capturing real-world scenarios, especially rare edge cases such as near-accidents or unusual road conditions, is inherently difficult. Synthetic data allows researchers to simulate these scenarios in a controlled virtual environment, ensuring that models are trained to handle a wide range of conditions. This approach has proven invaluable for advancing the capabilities of self-driving cars.

Another important application of synthetic data lies in augmenting existing datasets. Introducing variations into datasets enhances model robustness by exposing the model to diverse conditions. For instance, in speech recognition, data augmentation techniques like SpecAugment (Park et al. 2019) introduce noise, shifts, or pitch variations, enabling models to generalize better across different environments and speaker styles. This principle extends to other do-

[4] Diffusion models use noise prediction across time to simulate generation, while flow-matching algorithms minimize the displacement between source and target distributions.

[5] Generative Adversarial Networks (GANs): Machine learning models with a generator creating data and a discriminator assessing its realism.

[6] Variational Autoencoders (VAEs): Generative models that encode data into a latent space and decode it to generate new samples.

mains as well, where synthetic data can fill gaps in underrepresented scenarios or edge cases.

In addition to expanding datasets, synthetic data addresses critical ethical and privacy concerns. Unlike real-world data, synthetic data attempts to not tie back to specific individuals or entities. This makes it especially useful in sensitive domains such as finance, healthcare, or human resources, where data confidentiality is paramount. The ability to preserve statistical properties while removing identifying information allows researchers to maintain high ethical standards without compromising the quality of their models. In healthcare, privacy regulations such as GDPR[7] and HIPAA[8] limit the sharing of sensitive patient information. Synthetic data generation enables the creation of realistic yet anonymized datasets that can be used for training diagnostic models without compromising patient privacy.

Poorly generated data can misrepresent underlying real-world distributions, introducing biases or inaccuracies that degrade model performance. Validating synthetic data against real-world benchmarks is essential to ensure its reliability. Additionally, models trained primarily on synthetic data must be rigorously tested in real-world scenarios to confirm their ability to generalize effectively. Another challenge is the potential amplification of biases present in the original datasets used to inform synthetic data generation. If these biases are not carefully addressed, they may be inadvertently reinforced in the resulting models. A critical consideration is maintaining proper balance between synthetic and real-world data during training - if models are overly trained on synthetic data, their outputs may become nonsensical and model performance may collapse.

Synthetic data has revolutionized the way machine learning systems are trained, providing flexibility, diversity, and scalability in data preparation. However, as its adoption grows, practitioners must remain vigilant about its limitations and ethical implications. By combining synthetic data with rigorous validation and thoughtful application, machine learning researchers and engineers can unlock its full potential while ensuring reliability and fairness in their systems.

6.4.6 Continuing the KWS Example

KWS is an excellent case study of how different data collection approaches can be combined effectively. Each method we've discussed plays a role in building robust wake word detection systems, albeit with different trade-offs:

Pre-existing datasets like Google's Speech Commands (Warden 2018) provide a foundation for initial development, offering carefully curated voice samples for common wake words. However, these datasets often lack diversity in accents, environments, and languages, necessitating additional data collection strategies.

Web scraping can supplement these baseline datasets by gathering diverse voice samples from video platforms, podcast repositories, and speech databases. This helps capture natural speech patterns and wake word variations, though careful attention must be paid to audio quality and privacy considerations when scraping voice data.

[7] GDPR: General Data Protection Regulation, a legal framework that sets guidelines for the collection and processing of personal information in the EU.

[8] HIPAA: Health Insurance Portability and Accountability Act, U.S. legislation that provides data privacy and security provisions for safeguarding medical information.

Crowdsourcing becomes valuable for collecting specific wake word samples across different demographics and environments. Platforms like Amazon Mechanical Turk can engage contributors to record wake words in various accents, speaking styles, and background conditions. This approach is particularly useful for gathering data for underrepresented languages or specific acoustic environments.

Synthetic data generation helps fill remaining gaps by creating unlimited variations of wake word utterances. Using speech synthesis (Werchniak et al. 2021) and audio augmentation techniques, developers can generate training data that captures different acoustic environments (busy streets, quiet rooms, moving vehicles), speaker characteristics (age, accent, gender), and background noise conditions.

This multi-faceted approach to data collection enables the development of KWS systems that perform robustly across diverse real-world conditions. The combination of methods helps address the unique challenges of wake word detection, from handling various accents and background noise to maintaining consistent performance across different devices and environments.

6.5 Data Ingestion

The collected data must be reliably and efficiently ingested into our ML systems through well-designed data pipelines. This transformation presents several challenges that ML engineers must address.

6.5.1 Ingestion Patterns

In ML systems, data ingestion typically follows two primary patterns: batch ingestion and stream ingestion. Each pattern has distinct characteristics and use cases that students should understand to design effective ML systems.

Batch ingestion involves collecting data in groups or batches over a specified period before processing. This method is appropriate when real-time data processing is not critical and data can be processed at scheduled intervals. It's also useful for loading large volumes of historical data. For example, a retail company might use batch ingestion to process daily sales data overnight, updating their ML models for inventory prediction each morning (Akidau et al. 2015).

In contrast, stream ingestion processes data in real-time as it arrives. This pattern is crucial for applications requiring immediate data processing, scenarios where data loses value quickly, and systems that need to respond to events as they occur. A financial institution, for instance, might use stream ingestion for real-time fraud detection, processing each transaction as it occurs to flag suspicious activity immediately (Kleppmann 2016).

Many modern ML systems employ a hybrid approach, combining both batch and stream ingestion to handle different data velocities and use cases. This flexibility allows systems to process both historical data in batches and real-time data streams, providing a comprehensive view of the data landscape.

6.5.2 ETL and ELT Comparison

When designing data ingestion pipelines for ML systems, it's necessary to understand the differences between Extract, Transform, Load (ETL) and Extract, Load, Transform (ELT) approaches, as illustrated in Figure 6.9. These paradigms determine when data transformations occur relative to the loading phase, significantly impacting the flexibility and efficiency of your ML pipeline.

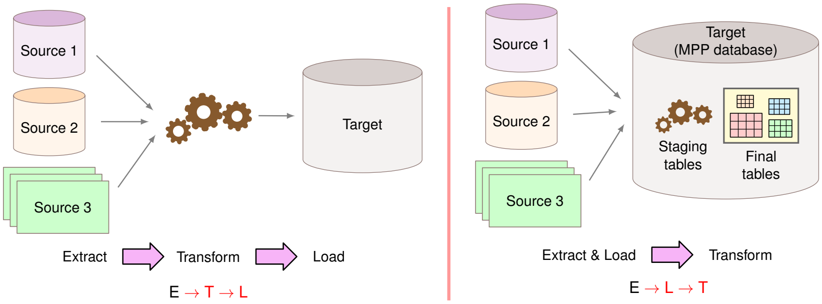


Figure 6.9: Key differences between Extract, Transform, Load (ETL) versus Extract, Load, Transform (ELT).

ETL is a well-established paradigm in which data is first gathered from a source, then transformed to match the target schema or model, and finally loaded into a data warehouse or other repository. This approach typically results in data being stored in a ready-to-query format, which can be advantageous for ML systems that require consistent, pre-processed data. For instance, an ML system predicting customer churn might use ETL to standardize and aggregate customer interaction data from multiple sources before loading it into a format suitable for model training (Inmon 2005).

However, ETL can be less flexible when schemas or requirements change frequently, a common occurrence in evolving ML projects. This is where the ELT approach comes into play. ELT reverses the order by first loading raw data and then applying transformations as needed. This method is often seen in modern data lake or schema-on-read[9] environments, allowing for a more agile approach when addressing evolving analytical needs in ML systems.

By deferring transformations, ELT can accommodate varying uses of the same dataset, which is particularly useful in exploratory data analysis phases of ML projects or when multiple models with different data requirements are being developed simultaneously. However, it's important to note that ELT places greater demands on storage systems and query engines, which must handle large amounts of unprocessed information.

In practice, many ML systems employ a hybrid approach, selecting ETL or ELT on a case-by-case basis depending on the specific requirements of each data source or ML model. For example, a system might use ETL for structured data from relational databases where schemas are well-defined and stable, while employing ELT for unstructured data like text or images where transformation requirements may evolve as the ML models are refined.

[9] Schema-on-read: A flexible approach where data structure is defined at access time, not during ingestion, enabling versatile use of raw data.

6.5.3 Data Source Integration

Integrating diverse data sources is a key challenge in data ingestion for ML systems. Data may come from various origins, including databases, APIs, file systems, and IoT devices. Each source may have its own data format, access protocol, and update frequency.

To effectively integrate these sources, ML engineers must develop robust connectors or adapters for each data source. These connectors handle the specifics of data extraction, including authentication, rate limiting, and error handling. For example, when integrating with a REST API, the connector would manage API keys, respect rate limits, and handle HTTP status codes appropriately.

Furthermore, source integration often involves data transformation at the ingestion point. This might include parsing JSON or XML responses, converting timestamps to a standard format, or performing basic data cleaning operations. The goal is to standardize the data format as it enters the ML pipeline, simplifying downstream processing.

It's also essential to consider the reliability and availability of data sources. Some sources may experience downtime or have inconsistent data quality. Implementing retry mechanisms, data quality checks, and fallback procedures can help ensure a steady flow of reliable data into the ML system.

6.5.4 Validation Techniques

Data validation is an important step in the ingestion process, ensuring that incoming data meets quality standards and conforms to expected schemas. This step helps prevent downstream issues in ML pipelines caused by data anomalies or inconsistencies.

At the ingestion stage, validation typically encompasses several key aspects. First, it checks for schema conformity, ensuring that incoming data adheres to the expected structure, including data types and field names. Next, it verifies data ranges and constraints, confirming that numeric fields fall within expected ranges and that categorical fields contain valid values. Completeness checks are also performed, looking for missing or null values in required fields. Additionally, consistency checks ensure that related data points are logically coherent (Gudivada, Rao, et al. 2017).

For example, in a healthcare ML system ingesting patient data, validation might include checking that age values are positive integers, diagnosis codes are from a predefined set, and admission dates are not in the future. By implementing robust validation at the ingestion stage, ML engineers can detect and handle data quality issues early, significantly reducing the risk of training models on flawed or inconsistent data.

6.5.5 Error Management

Error handling in data ingestion is essential for building resilient ML systems. Errors can occur at various points in the ingestion process, from source connection issues to data validation failures. Effective error handling strategies ensure that the ML pipeline can continue to operate even when faced with data ingestion challenges.

A key concept in error handling is graceful degradation. This involves designing systems to continue functioning, possibly with reduced capabilities, when faced with partial data loss or temporary source unavailability. Implementing intelligent retry logic for transient errors, such as network interruptions or temporary service outages, is another important aspect of robust error handling. Many ML systems employ the concept of dead letter queues[10], using separate storage for data that fails processing. This allows for later analysis and potential reprocessing of problematic data (Kleppmann 2016).

For instance, in a financial ML system ingesting market data, error handling might involve falling back to slightly delayed data sources if real-time feeds fail, while simultaneously alerting the operations team to the issue. This approach ensures that the system continues to function and that responsible parties are aware of and can address the problem.

This ensures that downstream processes have access to reliable, high-quality data for training and inference tasks, even in the face of ingestion challenges. Understanding these concepts of data validation and error handling is essential for students and practitioners aiming to build robust, production-ready ML systems.

Once ingestion is complete and data is validated, it is typically loaded into a storage environment suited to the organization's analytical or machine learning needs. Some datasets flow into data warehouses for structured queries, whereas others are retained in data lakes for exploratory or large-scale analyses. Advanced systems may also employ feature stores to provide standardized features for machine learning.

6.5.6 Continuing the KWS Example

A production KWS system typically employs both streaming and batch ingestion patterns. The streaming pattern handles real-time audio data from active devices, where wake words must be detected with minimal latency. This requires careful implementation of pub/sub mechanisms—for example, using Apache Kafka-like streams to buffer incoming audio data and enable parallel processing across multiple inference servers.

Simultaneously, the system processes batch data for model training and updates. This includes ingesting new wake word recordings from crowdsourcing efforts, synthetic data from voice generation systems, and validated user interactions. The batch processing typically follows an ETL pattern, where audio data is preprocessed (normalized, filtered, segmented) before being stored in a format optimized for model training.

KWS systems must integrate data from diverse sources, such as real-time audio streams from deployed devices, crowdsourced recordings from data collection platforms etc. Each source presents unique challenges. Real-time audio streams require rate limiting to prevent system overload during usage spikes. Crowdsourced data needs robust validation to ensure recording quality and correct labeling. Synthetic data must be verified for realistic representation of wake word variations.

KWS systems employ sophisticated error handling mechanisms due to the nature of voice interaction. When processing real-time audio, dead letter queues

[10] Dead Letter Queues: Queues that store unprocessed messages for analysis or reprocessing.

store failed recognition attempts for analysis, helping identify patterns in false negatives or system failures. Data validation becomes particularly important for maintaining wake word detection accuracy—incoming audio must be checked for quality issues like clipping, noise levels, and appropriate sampling rates.

For example, consider a smart home device processing the wake word "Alexa." The ingestion pipeline must validate:

- Audio quality metrics (signal-to-noise ratio, sample rate, bit depth)
- Recording duration (typically 1-2 seconds for wake words)
- Background noise levels
- Speaker proximity indicators

Invalid samples are routed to dead letter queues for analysis, while valid samples are processed in real-time for wake word detection.

This case study illustrates how real-world ML systems must carefully balance different ingestion patterns, handle multiple data sources, and maintain robust error handling—all while meeting strict latency and reliability requirements. The lessons from KWS systems apply broadly to other ML applications requiring real-time processing capabilities alongside continuous model improvement.

6.6 Data Processing

Data processing is a stage in the machine learning pipeline that transforms raw data into a format suitable for model training and inference. This stage encompasses several key activities, each playing a role in preparing data for effective use in ML systems. The approach to data processing is closely tied to the ETL (Extract, Transform, Load) or ELT (Extract, Load, Transform) paradigms discussed earlier.

In traditional ETL workflows, much of the data processing occurs before the data is loaded into the target system. This approach front-loads the cleaning, transformation, and feature engineering steps, ensuring that data is in a ready-to-use state when it reaches the data warehouse or ML pipeline. ETL is often preferred when dealing with structured data or when there's a need for significant data cleansing before analysis.

Conversely, in ELT workflows, raw data is first loaded into the target system, and transformations are applied afterwards. This approach, often used with data lakes, allows for more flexibility in data processing. It's particularly useful when dealing with unstructured or semi-structured data, or when the exact transformations needed are not known in advance. In ELT, many of the data processing steps we'll discuss might be performed on-demand or as part of the ML pipeline itself.

The choice between ETL and ELT can impact how and when data processing occurs in an ML system. For instance, in an ETL-based system, data cleaning and initial transformations might happen before the data even reaches the ML team. In contrast, an ELT-based system might require ML engineers to handle more of the data processing tasks as part of their workflow.

Regardless of whether an organization follows an ETL or ELT approach, understanding the following data processing steps is crucial for ML practitioners.

These processes ensure that data is clean, relevant, and optimally formatted for machine learning algorithms.

6.6.1 Cleaning Techniques

Data cleaning involves identifying and correcting errors, inconsistencies, and inaccuracies in datasets. Raw data frequently contains issues such as missing values, duplicates, or outliers that can significantly impact model performance if left unaddressed.

In practice, data cleaning might involve removing duplicate records, handling missing values through imputation or deletion, and correcting formatting inconsistencies. For instance, in a customer database, names might be inconsistently capitalized or formatted. A data cleaning process would standardize these entries, ensuring that “John Doe,” “john doe,” and “DOE, John” are all treated as the same entity.

Outlier detection and treatment is another important aspect of data cleaning. Outliers can sometimes represent valuable information about rare events, but they can also be the result of measurement errors or data corruption. ML practitioners must carefully consider the nature of their data and the requirements of their models when deciding how to handle outliers.

6.6.2 Data Quality Assessment

Quality assessment goes hand in hand with data cleaning, providing a systematic approach to evaluating the reliability and usefulness of data. This process involves examining various aspects of data quality, including accuracy, completeness, consistency, and timeliness.

Tools and techniques for quality assessment range from simple statistical measures to more complex machine learning-based approaches. For example, data profiling tools can provide summary statistics and visualizations that help identify potential quality issues. More advanced techniques might involve using unsupervised learning algorithms to detect anomalies or inconsistencies in large datasets.

Establishing clear quality metrics and thresholds is essential for maintaining data quality over time. These metrics might include the percentage of missing values, the frequency of outliers, or measures of data freshness. Regular quality assessments help ensure that data entering the ML pipeline meets the necessary standards for reliable model training and inference.

6.6.3 Transformation Techniques

Data transformation converts the data from its raw form into a format more suitable for analysis and modeling. This process can include a wide range of operations, from simple conversions to complex mathematical transformations.

Common transformation tasks include normalization and standardization, which scale numerical features to a common range or distribution. For example, in a housing price prediction model, features like square footage and number of rooms might be on vastly different scales. Normalizing these features ensures that they contribute more equally to the model’s predictions (Bishop 2006).

Other transformations might involve encoding categorical variables, handling date and time data, or creating derived features. For instance, one-hot encoding[11] is often used to convert categorical variables into a format that can be readily understood by many machine learning algorithms.

6.6.4 Feature Engineering

Feature engineering is the process of using domain knowledge to create new features that make machine learning algorithms work more effectively. This step is often considered more of an art than a science, requiring creativity and deep understanding of both the data and the problem at hand.

Feature engineering might involve combining existing features, extracting information from complex data types, or creating entirely new features based on domain insights. For example, in a retail recommendation system, engineers might create features that capture the recency, frequency, and monetary value of customer purchases, known as RFM analysis (Kuhn and Johnson 2013).

The importance of feature engineering cannot be overstated. Well-engineered features can often lead to significant improvements in model performance, sometimes outweighing the impact of algorithm selection or hyperparameter tuning.

6.6.5 Processing Pipeline Design

Processing pipelines bring together the various data processing steps into a coherent, reproducible workflow. These pipelines ensure that data is consistently prepared across training and inference stages, reducing the risk of data leakage and improving the reliability of ML systems.

Modern ML frameworks and tools often provide capabilities for building and managing data processing pipelines. For instance, Apache Beam and TensorFlow Transform allow developers to define data processing steps that can be applied consistently during both model training and serving.

Effective pipeline design involves considerations such as modularity, scalability, and version control. Modular pipelines allow for easy updates and maintenance of individual processing steps. Version control for pipelines is crucial, ensuring that changes in data processing can be tracked and correlated with changes in model performance. This modular breakdown of pipeline components is well illustrated by TensorFlow Extended in Figure 6.10, which shows the complete flow from initial data ingestion through to final model deployment.

6.6.6 Scalability Considerations

As datasets grow larger and ML systems become more complex, the scalability of data processing becomes increasingly important. Processing large volumes of data efficiently often requires distributed computing approaches and careful consideration of computational resources.

Techniques for scaling data processing include parallel processing, where data is divided across multiple machines or processors for simultaneous processing. Distributed frameworks like Apache Spark are commonly used for this

[11] One-Hot Encoding: Converts categorical variables into binary vectors, where each category is represented by a unique vector with one element set to 1 and the rest to 0. This allows categorical data to be used in ML models requiring numerical input.

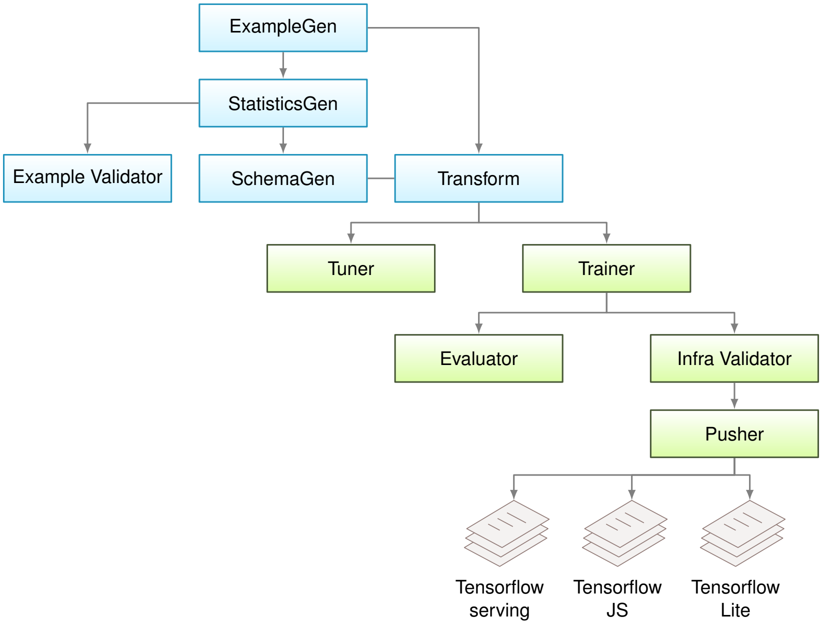


Figure 6.10: Example breakdown of the entire data to model processing pipeline for TensorFlow Extended.

purpose, allowing data processing tasks to be scaled across large clusters of computers.

Another important consideration is the balance between preprocessing and on-the-fly computation. While extensive preprocessing can speed up model training and inference, it can also lead to increased storage requirements and potential data staleness. Some ML systems opt for a hybrid approach, preprocessing certain features while computing others on-the-fly during model training or inference.

Effective data processing is fundamental to the success of ML systems. By carefully cleaning, transforming, and engineering data, practitioners can significantly improve the performance and reliability of their models. As the field of machine learning continues to evolve, so too do the techniques and tools for data processing, making this an exciting and dynamic area of study and practice.

6.6.7 Continuing the KWS Example

A KWS system requires careful cleaning of audio recordings to ensure reliable wake word detection. Raw audio data often contains various imperfections—background noise, clipped signals, varying volumes, and inconsistent sampling rates. For example, when processing the wake word "Alexa," the system must clean recordings to standardize volume levels, remove ambient noise, and ensure consistent audio quality across different recording environments, all while preserving the essential characteristics that make the wake word recognizable.

Building on clean data, quality assessment becomes important for KWS systems. Quality metrics for KWS data are uniquely focused on audio characteristics, including signal-to-noise ratio (SNR), audio clarity scores, and speaking rate consistency. For instance, a KWS quality assessment pipeline might automatically flag recordings where background noise exceeds acceptable thresholds or where the wake word is spoken too quickly or unclearly, ensuring only high-quality samples are used for model training.

These quality metrics must be carefully calibrated to reflect real-world operating conditions. A robust training dataset incorporates both pristine recordings and samples containing controlled levels of environmental variations. For instance, while recordings with signal-masking interference are excluded, the dataset should include samples with measured background acoustics, variable speaker distances, and concurrent speech or other forms of audio signals. This approach to data diversity ensures the model maintains wake word detection reliability across the full spectrum of deployment environments and acoustic conditions.

Once quality is assured, transforming audio data for KWS involves converting raw waveforms into formats suitable for ML models. The typical transformation pipeline converts audio signals into spectrograms[12] or mel-frequency cepstral coefficients (MFCCs)[13], standardizing the representation across different recording conditions. This transformation must be consistently applied across both training and inference, often with additional considerations for real-time processing on edge devices.

Figure 6.11 illustrates this transformation process. The top panel is a raw waveform of a simulated audio signal, which consists of a sine wave mixed with noise. This time-domain representation highlights the challenges posed by real-world recordings, where noise and variability must be addressed. The middle panel shows the spectrogram of the signal, which maps its frequency content over time. The spectrogram provides a detailed view of how energy is distributed across frequencies, making it easier to analyze patterns that could influence wake word recognition, such as the presence of background noise or signal distortions The bottom panel shows the MFCCs, derived from the spectrogram. These coefficients compress the audio information into a format that emphasizes speech-related characteristics, making them well-suited for KWS tasks.

With transformed data in hand, feature engineering for KWS focuses on extracting characteristics that help distinguish wake words from background speech. Engineers might create features capturing tonal variations, speech energy patterns, or temporal characteristics. For the wake word “Alexa,” features might include energy distribution across frequency bands, pitch contours, and duration patterns that characterize typical pronunciations. While hand-engineered speech features have seen much success, learned features (Zeghidour et al. 2021) are increasingly common.

In practice, bringing all these elements together, KWS processing pipelines must handle both batch processing for training and real-time processing for inference. The pipeline typically includes stages for audio preprocessing, feature extraction, and quality filtering. Importantly, these pipelines must be designed to operate efficiently on edge devices while maintaining consistent processing steps between training and deployment.

[12] Spectrogram: A visual representation of the spectrum of frequencies in a signal as it varies over time, commonly used in audio processing.

[13] Mel-Frequency Cepstral Coefficients (MFCCs): Features extracted from audio signals that represent the short-term power spectrum, widely used in speech and audio analysis.

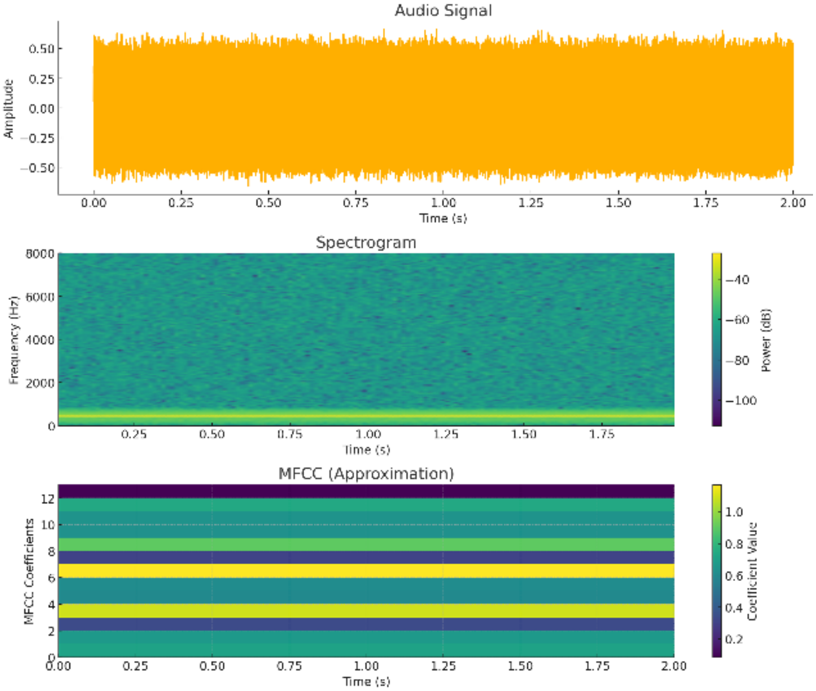


Figure 6.11: KWS data processing of an audio signal (top panel) represented in a spectrogram (middle panel) showing the energy distribution across time and frequency, along with the corresponding MFCCs (bottom panel) that capture perceptually relevant features.

6.7 Data Labeling

While data engineering encompasses many aspects of preparing data for machine learning systems, data labeling represents a particularly complex systems challenge. As training datasets grow to millions or billions of examples,[14] the infrastructure supporting labeling operations becomes increasingly critical to system performance.

Modern machine learning systems must efficiently handle the creation, storage, and management of labels across their data pipeline. The systems architecture must support various labeling workflows while maintaining data consistency, ensuring quality, and managing computational resources effectively. These requirements compound when dealing with large-scale datasets or real-time labeling needs.

The systematic challenges extend beyond just storing and managing labels. Production ML systems need robust pipelines that integrate labeling workflows with data ingestion, preprocessing, and training components. These pipelines must maintain high throughput while ensuring label quality and adapting to changing requirements. For instance, a speech recognition system might need to continuously update its training data with new audio samples and corresponding transcription labels, requiring careful coordination between data collection, labeling, and training subsystems.

Infrastructure requirements vary significantly based on labeling approach and scale. Manual expert labeling may require specialized interfaces and se-

[14] Modern ML models often contain millions or billions of parameters to capture complex data patterns. A general rule of thumb in ML is to have a training dataset that is at least 10 times larger than the model's parameter count to ensure robust learning and avoid overfitting.

curity controls, while automated labeling systems need substantial compute resources for inference. Organizations must carefully balance these requirements against performance needs and resource constraints.

We explore how data labeling fundamentally shapes machine learning system design. From storage architectures to quality control pipelines, each aspect of the labeling process introduces unique technical challenges that ripple throughout the ML infrastructure. Understanding these systems-level implications is essential for building robust, scalable labeling solutions which are an integral part of data negineering.

6.7.1 Types of Labels

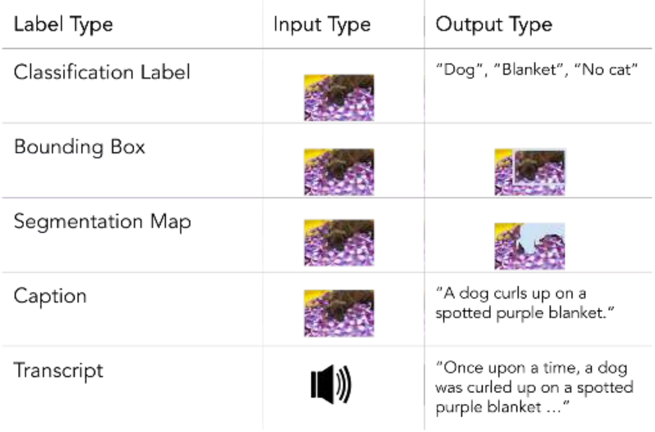
To build effective machine learning systems, we must first understand how different types of labels affect our system architecture and resource requirements. Let's explore this through a practical example: imagine building a smart city system that needs to detect and track various objects like vehicles, pedestrians, and traffic signs from video feeds. Labels capture information about key tasks or concepts.

- **Classification labels** are the simplest form, categorizing images with a specific tag or (in multi-label classification) tags (e.g., labeling an image as "car" or "pedestrian"). While conceptually straightforward, a production system processing millions of video frames must efficiently store and retrieve these labels.
- **Bounding boxes** go further by identifying object locations, drawing a box around each object of interest. Our system now needs to track not just what objects exist, but where they are in each frame. This spatial information introduces new storage and processing challenges, especially when tracking moving objects across video frames.
- **Segmentation maps** provide the most detailed information by classifying objects at the pixel level, highlighting each object in a distinct color. For our traffic monitoring system, this might mean precisely outlining each vehicle, pedestrian, and road sign. These detailed annotations significantly increase our storage and processing requirements.

Figure 6.12 illustrates the common label types:

The choice of label format depends heavily on our system requirements and resource constraints (Johnson-Roberson et al. 2017). While classification labels might suffice for simple traffic counting, autonomous vehicles need detailed segmentation maps to make precise navigation decisions. Leading autonomous vehicle companies often maintain hybrid systems that store multiple label types for the same data, allowing flexible use across different applications.

Beyond the core labels, production systems must also handle rich metadata. The Common Voice dataset (Ardila et al. 2020), for instance, exemplifies this in its management of audio data for speech recognition. The system tracks speaker demographics for model fairness, recording quality metrics for data filtering, validation status for label reliability, and language information for multilingual support.



Label Type	Input Type	Output Type
Classification Label		"Dog", "Blanket", "No cat"
Bounding Box		
Segmentation Map		
Caption		"A dog curls up on a spotted purple blanket."
Transcript		"Once upon a time, a dog was curled up on a spotted purple blanket ..."

Figure 6.12: An overview of common label types.

Modern labeling platforms have built sophisticated metadata management systems to handle these complex relationships. This metadata becomes important for maintaining and managing data quality and debugging model behavior. If our traffic monitoring system performs poorly in rainy conditions, having metadata about weather conditions during data collection helps identify and address the issue. The infrastructure must efficiently index and query this metadata alongside the primary labels.

The choice of label type cascades through our entire system design. A system built for simple classification labels would need significant modifications to handle segmentation maps efficiently. The infrastructure must optimize storage systems for the chosen label format, implement efficient data retrieval patterns for training, maintain quality control pipelines for validation, and manage version control for label updates. Resource allocation becomes particularly critical as data volume grows, requiring careful capacity planning across storage, compute, and networking components.

6.7.2 Annotation Techniques

Manual labeling by experts is the primary approach in many annotation pipelines. This method produces high-quality results but also raises considerable system design challenges. For instance, in medical imaging systems, experienced radiologists offer essential annotations. Such systems necessitate specialized interfaces for accurate labeling, secure data access controls to protect patient privacy, and reliable version control mechanisms to monitor annotation revisions. Despite the dependable outcomes of expert labeling, the scarcity and high expenses of specialists render it challenging to implement on a large scale for extensive datasets.

As we discussed earlier, crowdsourcing offers a path to greater scalability by distributing annotation tasks across many annotators (Sheng and Zhang 2019). Crowdsourcing enables non-experts to distribute annotation tasks, often through dedicated platforms (Sheng and Zhang 2019). Several companies have emerged as leaders in this space, building sophisticated platforms for large-scale annotation. For instance, companies such as Scale AI specialize in managing thousands of concurrent annotators through their platform. Appen focuses on linguistic annotation and text data, while Labelbox has developed specialized tools for computer vision tasks. These platforms allow dataset creators to access a large pool of annotators, making it possible to label vast amounts of data relatively quickly.

Weakly supervised and programmatic methods represent a third approach, using automation to reduce manual effort (Ratner et al. 2018). These systems leverage existing knowledge bases and heuristics to automatically generate labels. For example, distant supervision techniques might use a knowledge base to label mentions of companies in text data. While these methods can rapidly label large datasets, they require substantial compute resources for inference, sophisticated caching systems to avoid redundant computation, and careful monitoring to manage potential noise and bias.

Most production systems combine multiple annotation approaches to balance speed, cost, and quality. A common pattern employs programmatic labeling for initial coverage, followed by crowdsourced verification and expert review of uncertain cases. This hybrid approach requires careful system design to manage the flow of data between different annotation stages. The infrastructure must track label provenance, manage quality control at each stage, and ensure consistent data access patterns across different annotator types.

The choice of annotation method significantly impacts system architecture. Expert-only systems might employ centralized architectures with high-speed access to a single data store. Crowdsourcing demands distributed architectures to handle concurrent annotators. Automated systems need substantial compute resources and caching infrastructure. Many organizations implement tiered architectures where different annotation methods operate on different subsets of data based on complexity and criticality.

Clear guidelines and thorough training remain essential regardless of the chosen architecture. The system must provide consistent interfaces, documentation, and quality metrics across all annotation methods. This becomes particularly challenging when managing diverse annotator pools with varying levels of expertise. Some platforms address this by offering access to specialized annotators. For instance, providing medical professionals for healthcare datasets or domain experts for technical content.

6.7.3 Label Quality Assessment

Label quality is extremely important for machine learning system performance. A model can only be as good as its training data. However, ensuring quality at scale presents significant systems challenges. The fundamental challenge stems from label uncertainty.

Figure 6.13 illustrates common failure modes in labeling systems: some errors arise from data quality issues (like the blurred frog image), while others require deep domain expertise (as with the black stork identification). Even with clear instructions and careful system design, some fraction of labels will inevitably be incorrect Thyagarajan et al. (2022).

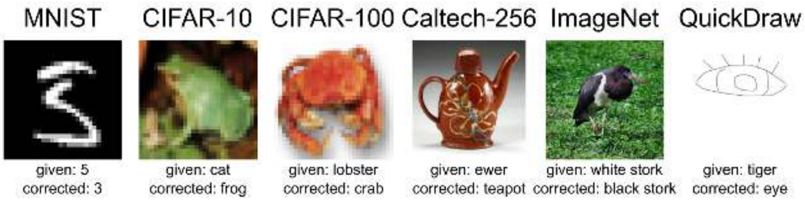


Figure 6.13: Some examples of hard labeling cases. Source: Northcutt, Athalye, and Mueller (2021)

Production ML systems implement multiple layers of quality control to address these challenges. Typically, systematic quality checks continuously monitor the labeling pipeline. These systems randomly sample labeled data for expert review and employ statistical methods to flag potential errors. The infrastructure must efficiently process these checks across millions of examples without creating bottlenecks in the labeling pipeline.

Collecting multiple labels per data point, often referred to as "consensus labeling," can help identify controversial or ambiguous cases. Professional labeling companies have developed sophisticated infrastructure for this process. For example, Labelbox has consensus tools that track inter-annotator agreement rates and automatically route controversial cases for expert review. Scale AI implements tiered quality control, where experienced annotators verify the work of newer team members.

Beyond technical infrastructure, successful labeling systems must consider human factors. When working with annotators, organizations need robust systems for training and guidance. This includes good documentation, clear examples of correct labeling, and regular feedback mechanisms. For complex or domain-specific tasks, the system might implement tiered access levels, routing challenging cases to annotators with appropriate expertise.

Ethical considerations also significantly impact system design. For datasets containing potentially disturbing content, systems should implement protective features like grayscale viewing options (Blackwood et al. 2019). This requires additional image processing pipelines and careful interface design. We need to develop workload management systems that track annotator exposure to sensitive content and enforce appropriate limits.

The quality control system itself generates substantial data that must be efficiently processed and monitored. Organizations typically track inter-annotator agreement rates, label confidence scores, time spent per annotation, error patterns and types, annotator performance metrics, and bias indicators. These metrics must be computed and updated efficiently across millions of examples, often requiring dedicated analytics pipelines.

Regular bias audits are another critical component of quality control. Systems must monitor for cultural, personal, or professional biases that could skew the labeled dataset. This requires infrastructure for collecting and analyzing demographic information, measuring label distributions across different annotator groups, identifying systematic biases in the labeling process, and implementing corrective measures when biases are detected.

Perhaps the most important aspect is that the process must remain iterative. As new challenges emerge, quality control systems must adapt and evolve.

Through careful system design and implementation of these quality control mechanisms, organizations can maintain high label quality even at a massive scale.

6.7.4 AI in Annotation

As machine learning systems grow in scale and complexity, organizations increasingly leverage AI to accelerate and enhance their labeling pipelines. This approach introduces new system design considerations around model deployment, resource management, and human-AI collaboration. The fundamental challenge stems from data volume. Manual annotation alone cannot keep pace with modern ML systems' data needs. As illustrated in Figure 6.14, AI assistance offers several paths to scale labeling operations, each requiring careful system design to balance speed, quality, and resource usage.

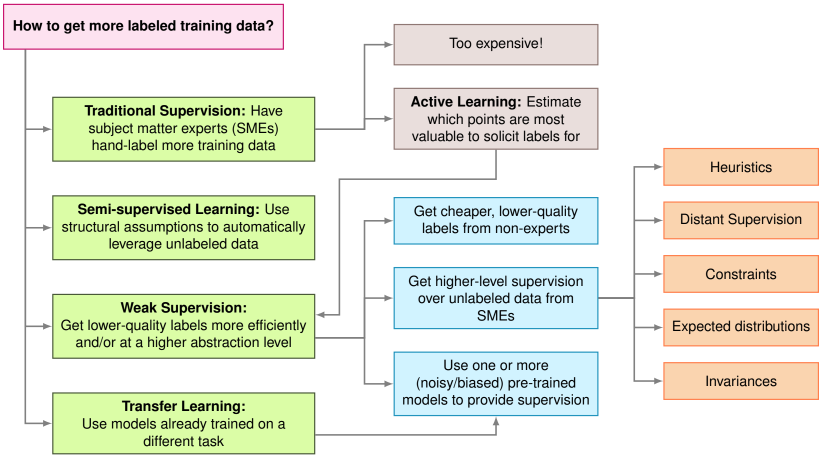

How to get more labeled training data?
Too expensive!
Traditional Supervision: Have subject matter experts (SMEs) hand-label more training data
Active Learning: Estimate which points are most valuable to solicit labels for
Semi-supervised Learning: Use structural assumptions to automatically leverage unlabeled data
Get cheaper, lower-quality labels from non-experts
Weak Supervision: Get lower-quality labels more efficiently and/or at a higher abstraction level
Get higher-level supervision over unlabeled data from SMEs
Use one or more (noisy/biased) pre-trained models to provide supervision
Transfer Learning: Use models already trained on a different task
Heuristics
Distant Supervision
Constraints
Expected distributions
Invariances

Figure 6.14: Strategies for acquiring additional labeled training data. Source: Stanford AI Lab.

Modern AI-assisted labeling typically employs a combination of approaches. Pre-annotation involves using AI models to generate preliminary labels for a dataset, which humans can then review and correct. Major labeling platforms have made significant investments in this technology. Snorkel AI uses programmatic labeling to automatically generate initial labels at scale. Scale AI deploys pre-trained models to accelerate annotation in specific domains like autonomous driving, while manycompanies like SuperAnnotate provide automated pre-labeling tools that can reduce manual effort drastically. This method, which often employs semi-supervised learning techniques (Chapelle, Scholkopf, and Zien 2009), can save a significant amount of time, especially for extremely large datasets.

The emergence of Large Language Models (LLMs) like ChatGPT has further transformed labeling pipelines. Beyond simple classification, LLMs can generate rich text descriptions, create labeling guidelines, and even explain their reasoning. For instance, content moderation systems use LLMs to perform initial content classification and generate explanations for policy violations. However, integrating LLMs introduces new system challenges around inference costs, rate limiting, and output validation. Many organizations adopt a tiered approach, using smaller specialized models for routine cases while reserving larger LLMs for complex scenarios.

Methods such as active learning[15] complement these approaches by intelligently prioritizing which examples need human attention (Coleman et al. 2022). These systems continuously analyze model uncertainty to identify valuable labeling candidates for humans to label. The infrastructure must efficiently compute uncertainty metrics, maintain task queues, and adapt prioritization strategies based on incoming labels. Consider a medical imaging system: active learning might identify unusual pathologies for expert review while handling routine cases automatically.

Quality control becomes increasingly crucial as these AI components interact. The system must monitor both AI and human performance, detect potential errors, and maintain clear label provenance. This requires dedicated infrastructure tracking metrics like model confidence and human-AI agreement rates. In safety-critical domains like self-driving cars, these systems must maintain particularly rigorous standards while processing massive streams of sensor data.

Real-world deployments demonstrate these principles at scale. Medical imaging systems (Krishnan, Rajpurkar, and Topol 2022) combine pre-annotation for common conditions with active learning for unusual cases, all while maintaining strict patient privacy.

Self-driving vehicle systems coordinate multiple AI models to label diverse sensor data in real-time. Social media platforms process millions of items hourly, using tiered approaches where simpler models handle clear cases while complex content routes to more sophisticated models or human reviewers.

While AI assistance offers clear benefits, it also introduces new failure modes. Systems must guard against bias amplification, where AI models trained on biased data perpetuate those biases in new labels. The infrastructure needs robust monitoring to detect such issues and mechanisms to break problematic feedback loops. Human oversight remains essential, requiring careful interface design to help annotators effectively supervise and correct AI output.

6.7.5 Labeling Challenges

While data labeling is essential for the development of supervised machine learning models, it comes with its own set of challenges and limitations that practitioners must be aware of and address. One of the primary challenges in data labeling is the inherent subjectivity in many labeling tasks. Even with clear guidelines, human annotators[16] may interpret data differently, leading to inconsistencies in labeling. This is particularly evident in tasks involving sentiment analysis, image classification of ambiguous objects, or labeling of complex medical conditions. For instance, in a study of medical image annotation, Oakden-Rayner et al. (2020) found significant variability in labels assigned by different radiologists, highlighting the challenge of obtaining "ground truth" in inherently subjective tasks.

Scalability presents another significant challenge, especially as datasets grow larger and more complex. Manual labeling is time-consuming and expensive,

[15] A machine learning approach where the model selects the most informative data points for labeling to improve learning efficiency.

[16] When involving human annotators in data labeling, organizations must protect individual privacy through robust anonymization, sanitization of identifying information, and secure data handling practices. Additionally, annotators themselves should be protected from exposure to potentially harmful content through appropriate content filtering and support systems.

often becoming a bottleneck in the machine learning pipeline. While crowdsourcing and AI-assisted methods can help address this issue to some extent, they introduce their own complications in terms of quality control and potential biases.

The issue of bias in data labeling is particularly concerning. Annotators bring their own cultural, personal, and professional biases to the labeling process, which can be reflected in the resulting dataset. For example, T. Wang et al. (2019) found that image datasets labeled predominantly by annotators from one geographic region showed biases in object recognition tasks, performing poorly on images from other regions. This highlights the need for diverse annotator pools and careful consideration of potential biases in the labeling process.

Data privacy and ethical considerations also pose challenges in data labeling. Leading data labeling companies have developed specialized solutions for these challenges. Scale AI, for instance, maintains dedicated teams and secure infrastructure for handling sensitive data in healthcare and finance. Appen implements strict data access controls and anonymization protocols, while Labelbox offers private cloud deployments for organizations with strict security requirements. When dealing with sensitive data, such as medical records or personal communications, ensuring annotator access while maintaining data privacy can be complex.

The dynamic nature of real-world data presents another limitation. Labels that are accurate at the time of annotation may become outdated or irrelevant as the underlying distribution of data changes over time. This concept, known as concept drift, necessitates ongoing labeling efforts and periodic re-evaluation of existing labels.

Lastly, the limitations of current labeling approaches become apparent when dealing with edge cases or rare events. In many real-world applications, it's the unusual or rare instances that are often most critical (e.g., rare diseases in medical diagnosis, or unusual road conditions in autonomous driving). However, these cases are, by definition, underrepresented in most datasets and may be overlooked or mislabeled in large-scale annotation efforts.

6.7.6 Continuing the KWS Example

The complex requirements of KWS reveal the role of automated data labeling in modern machine learning. The Multilingual Spoken Words Corpus (MSWC) (Mazumder et al. 2021) illustrates this through its innovative approach to generating labeled wake word data at scale. MSWC is large, containing over 23.4 million one-second spoken examples across 340,000 keywords in 50 different languages.

The core of this system, as illustrated in Figure 6.15, begins with paired sentence audio recordings and corresponding transcriptions, which can be sourced from projects like Common Voice or multilingual captioned content platforms such as YouTube. The system processes paired audio-text inputs through forced alignment[17] to identify word boundaries, extracts individual keywords as one-second segments, and generates a large-scale multilingual dataset suitable for training keyword spotting models. For example, when a speaker says, "He gazed up the steep bank," their voice generates a complex

[17] Forced Alignment: A technique in audio processing that synchronizes spoken words in an audio file with their corresponding text transcription by analyzing phoneme-level timing.

acoustic signal that conveys more than just the words themselves. This signal encapsulates subtle transitions between words, variations in pronunciation, and the natural rhythm of speech. The primary challenge lies in accurately pinpointing the exact location of each word within this continuous audio stream.

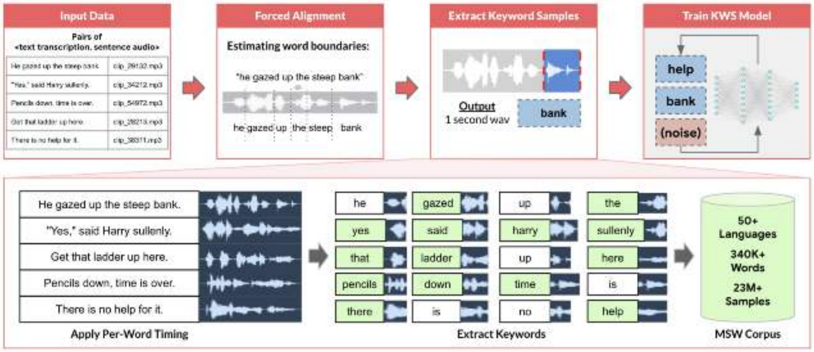

Input Data
Pairs of
«text transcription, sentence audio»
He gazed up the steep bank. clip_29132.mp3
"Yes," said Harry sullenly. clip_34212.mp3
Pencils down, time is over. clip_54972.mp3
Get that ladder up here. clip_28213.mp3
There is no help for it. clip_38311.mp3
Forced Alignment
Estimating word boundaries:
"he gazed up the steep bank"
he gazed up the steep bank
Extract Keyword Samples
Output
1 second wav
bank
Train KWS Model
help
bank
(noise)
He gazed up the steep bank.
"Yes," said Harry sullenly.
Get that ladder up here.
Pencils down, time is over.
There is no help for it.
Apply Per-Word Timing
he
gazed
up
the
yes
said
harry
sullenly
that
ladder
up
here
pencils
down
time
is
there
is
no
help
Extract Keywords
50+
Languages
340K+
Words
23M+
Samples
MSW Corpus

Figure 6.15: MSWC's automated data labeling pipeline.

This is where automated forced alignment proves useful. Tools such as the Montreal Forced Aligner (McAuliffe et al. 2017) analyze both the audio and its transcription, mapping the timing relationship between written words and spoken sounds, and attempts to mark the boundaries of when each word begins and ends in a speech recording at millisecond-level precision. For high-resource languages such as English, high-quality automated alignments are available "out-of-box" while alignments for low-resource languages must be bootstrapped on the speech data and transcriptions themselves, which can negatively impact timing quality.

With these precise timestamps, the extraction system can generate clean, one-second samples of individual keywords. However, this process requires careful engineering decisions. Background noise might interfere with detecting word boundaries. Speakers may stretch, compress, or mispronounce words in unexpected ways. Longer words may not fit within the default 1-second boundary. In order to aid ML practitioners in filtering out lower-quality samples in an automated fashion, MSWC provides a self-supervised anomaly detection algorithm, using acoustic embeddings to identify potential issues based on embedding distances to k-means clusters. This automated validation becomes particularly crucial given the scale of the dataset, which includes over 23 million samples across more than 340,000 words in 50+ languages. Traditional manual review could not maintain consistent standards across such volume without significant expense.

Modern voice assistant developers often build upon this type of labeling foundation. An automated corpus like MSWC may not contain the specific keywords an application developer wishes to use for their envisioned KWS system, but the corpus can provide a starting point for KWS prototyping in many underserved languages spoken around the world. While MSWC provides automated labeling at scale, production systems may add targeted human recording and verification for challenging cases, rare words, or difficult acoustic environments.

The infrastructure must gracefully coordinate between automated processing and human expertise.

The impact of this careful engineering extends far beyond the dataset itself. Automated labeling pipelines open new avenues to how we approach wake word detection and other ML tasks across languages or other demographic boundaries. Where manual collection and annotation might yield thousands of examples, automated dataset generation can yield millions while maintaining consistent quality. This enables voice interfaces to understand an ever-expanding vocabulary across the world's languages.

Through this approach to data labeling, MSWC demonstrates how thoughtful data engineering directly impacts production machine learning systems. The careful orchestration of forced alignment, extraction, and quality control creates a foundation for reliable voice interaction across languages. When a voice assistant responds to its wake word, it draws upon this sophisticated labeling infrastructure, which is a testament to the power of automated data processing in modern machine learning systems.

6.8 Data Storage

Machine learning workloads have data access patterns that differ markedly from those of traditional transactional systems or routine analytics. Whereas transactional databases optimize for frequent writes and row-level updates, most ML pipelines rely on high-throughput reads, large-scale data scans, and evolving schemas. This difference reflects the iterative nature of model development: data scientists repeatedly load and transform vast datasets to engineer features, test new hypotheses, and refine models.

Additionally, ML pipelines must accommodate real-world considerations such as evolving business requirements, new data sources, and changes in data availability. These realities push storage solutions to be both scalable and flexible, ensuring that organizations can manage data collected from diverse channels, ranging from sensor feeds to social media text, without constantly retooling the entire infrastructure. In this section, we will compare the practical use of databases, data warehouses, and data lakes for ML projects, then delve into how specialized services, metadata, and governance practices unify these varied systems into a coherent strategy.

6.8.1 Storage System Types

All raw and labeled data needs to be stored and accessed efficiently. When considering storage systems for ML, it is essential to understand the differences among different storage systems: databases, data warehouses, and data lakes. Each system has its strengths and is suited to different aspects of ML workflows.

Table 6.1 provides an overview of these storage systems. Databases usually support operational and transactional purposes. They work well for smaller, well-structured datasets, but can become cumbersome and expensive when applied to large-scale ML contexts involving unstructured data (such as images, audio, or free-form text).

Table 6.1: Comparative overview of the database, data warehouse, and data lake.

Attribute	Conventional Database	Data Warehouse	Data Lake
Purpose	Operational and transactional	Analytical and reporting	Storage for raw and diverse data for future processing
Data type	Structured	Structured	Structured, semi-structured, and unstructured
Scale	Small to medium volumes	Medium to large volumes	Large volumes of diverse data
Performance Optimization	Optimized for transactional queries (OLTP)	Optimized for analytical queries (OLAP)	Optimized for scalable storage and retrieval
Examples	MySQL, PostgreSQL, Oracle DB	Google BigQuery, Amazon Redshift, Microsoft Azure Synapse	Google Cloud Storage, AWS S3, Azure Data Lake Storage

Data warehouses, by contrast, are optimized for analytical queries across integrated datasets that have been transformed into a standardized schema. As indicated in the table, they handle large volumes of integrated data. Many ML systems successfully draw on data warehouses to power model training because the structured environment simplifies data exploration and feature engineering. Yet one limitation remains: a data warehouse may not accommodate truly unstructured data or rapidly changing data formats, particularly if the data originates from web scraping or Internet of Things (IoT) sensors.

Data lakes address this gap by storing structured, semi-structured, and unstructured data in its native format, deferring schema definitions until the point of reading or analysis (sometimes called *schema-on-read*)[18]. As Table 6.1 shows, data lakes can handle large volumes of diverse data types. This approach grants data scientists tremendous latitude when dealing with experimental use cases or novel data types. However, data lakes also demand careful cataloging and metadata management. Without sufficient governance, these expansive repositories risk devolving into unsearchable, disorganized silos.

The examples provided in Table 6.1 illustrate the range of technologies available for each storage system type. For instance, MySQL represents a traditional database system, while solutions like Google BigQuery and Amazon Redshift are examples of modern, cloud-based data warehouses. For data lakes, cloud storage solutions such as Google Cloud Storage, AWS S3, and Azure Data Lake Storage are commonly used due to their scalability and flexibility.

6.8.2 Storage Considerations

While traditional storage systems provide a foundation for ML workflows, the unique characteristics of machine learning workloads necessitate additional considerations. These ML-specific storage needs stem from the nature of ML development, training, and deployment processes, and addressing them is necessary for building efficient and scalable ML systems.

One of the primary challenges in ML storage is handling large model weights. Modern ML models, especially deep learning models, can have millions or even billions of parameters. For instance, GPT-3, a large language model, has 175 billion parameters, requiring approximately 350 GB of storage just for the model weights (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, and al. 2020). Storage

[18] Schema-on-read: A data management approach where data schema definitions are applied at the time of query or analysis rather than during initial data storage.

systems need to be capable of handling these large, often dense, numerical arrays efficiently, both in terms of storage capacity and access speed. This requirement goes beyond traditional data storage and enters the realm of high-performance computing storage solutions.

The iterative nature of ML development introduces another critical storage consideration: versioning for both datasets and models. Unlike traditional software version control, ML versioning needs to track large binary files efficiently. As data scientists experiment with different model architectures and hyperparameters, they generate numerous versions of models and datasets. Effective storage systems for ML must provide mechanisms to track these changes, revert to previous versions, and maintain reproducibility throughout the ML lifecycle. This capability is essential not only for development efficiency but also for regulatory compliance and model auditing in production environments.

Distributed training, often necessary for large models or datasets, generates substantial intermediate data, including partial model updates, gradients, and checkpoints. Storage systems for ML need to handle frequent, possibly concurrent, read and write operations of these intermediate results. Moreover, they should provide low-latency access to support efficient synchronization between distributed workers. This requirement pushes storage systems to balance between high throughput for large data transfers and low latency for quick synchronization operations.

The diversity of data types in ML workflows presents another unique challenge. ML systems often work with a wide variety of data, ranging from structured tabular data to unstructured images, audio, and text. Storage systems need to efficiently handle this diversity, often requiring a combination of different storage technologies optimized for specific data types. For instance, a single ML project might need to store and process tabular data in a columnar format for efficient feature extraction, while also managing large volumes of image data for computer vision tasks.

As organizations collect more data and create more sophisticated models, storage systems need to scale seamlessly. This scalability should support not just growing data volumes, but also increasing concurrent access from multiple data scientists and ML models. Cloud-based object storage systems have emerged as a popular solution due to their virtually unlimited scalability, but they introduce their own challenges in terms of data access latency and cost management.

The tension between sequential read performance for training and random access for inference is another key consideration. While training on large datasets benefits from high-throughput sequential reads, many ML serving scenarios require fast random access to individual data points or features. Storage systems for ML need to balance these potentially conflicting requirements, often leading to tiered storage architectures where frequently accessed data is kept in high-performance storage while less frequently used data is moved to cheaper, higher-latency storage.

The choice and configuration of storage systems can significantly impact the performance, cost-effectiveness, and overall success of ML initiatives. As the field of machine learning continues to evolve, storage solutions will need to adapt to meet the changing demands of increasingly sophisticated ML workflows.

6.8.3 Performance Factors

The performance of storage systems is critical in ML workflows, directly impacting the efficiency of model training, the responsiveness of inference, and the overall productivity of data science teams. Understanding and optimizing storage performance requires a focus on several key metrics and strategies tailored to ML workloads.

One of the primary performance metrics for ML storage is throughput, particularly for large-scale data processing and model training. High throughput is essential when ingesting and preprocessing vast datasets or when reading large batches of data during model training. For instance, distributed training of deep learning models on large datasets may require sustained read throughput of several gigabytes per second to keep GPU accelerators fully utilized.

Latency is another metric, especially for online inference and interactive data exploration. Low latency access to individual data points or small batches of data is vital for maintaining responsive ML services. In recommendation systems or real-time fraud detection, for example, storage systems must be able to retrieve relevant features or model parameters within milliseconds to meet strict service level agreements (SLAs).

The choice of file format can significantly impact both throughput and latency. Columnar storage formats such as Parquet or ORC[19] are particularly well-suited for ML workloads. These formats allow for efficient retrieval of specific features without reading entire records, substantially reducing I/O operations and speeding up data loading for model training and inference. For example, when training a model that only requires a subset of features from a large dataset, columnar formats can reduce data read times by an order of magnitude compared to row-based formats.

Compression is another key factor in storage performance optimization. While compression reduces storage costs and can improve read performance by reducing the amount of data transferred from disk, it also introduces computational overhead for decompression. The choice of compression algorithm often involves a trade-off between compression ratio and decompression speed. For ML workloads, fast decompression is usually prioritized over maximum compression, with algorithms like Snappy or LZ4 being popular choices.

Data partitioning strategies play a role in optimizing query performance for ML workloads. By intelligently partitioning data based on frequently used query parameters (such as date ranges or categorical variables), systems can dramatically improve the efficiency of data retrieval operations. For instance, in a recommendation system processing user interactions, partitioning data by user demographic attributes and time periods can significantly speed up the retrieval of relevant training data for personalized models.

To handle the scale of data in modern ML systems, distributed storage architectures are often employed. These systems, such as HDFS (Hadoop Distributed File System) or cloud-based object stores like Amazon S3, distribute data across multiple machines or data centers. This approach not only provides scalability but also enables parallel data access, which can substantially improve read performance for large-scale data processing tasks common in ML workflows.

[19] Parquet and ORC: Columnar storage formats optimized for analytical workloads and machine learning pipelines. They store data by columns rather than rows, enabling selective retrieval of specific features and reducing I/O overhead for large datasets.

Caching strategies are also vital for optimizing storage performance in ML systems. In-memory caching of frequently accessed data or computed features can significantly reduce latency and computational overhead. Distributed caching systems like Redis or Memcached are often used to scale caching capabilities across clusters of machines, providing low-latency access to hot data for distributed training or serving systems.

As ML workflows increasingly span from cloud to edge devices, storage performance considerations must extend to these distributed environments. Edge caching and intelligent data synchronization strategies become needed for maintaining performance in scenarios where network connectivity may be limited or unreliable. In the end, the goal is to create a storage infrastructure that can handle the volume and velocity of data in ML workflows while providing the low-latency access needed for responsive model training and inference.

6.8.4 Storage in ML Lifecycle

The storage needs of machine learning systems evolve significantly across different phases of the ML lifecycle. Understanding these changing requirements is important for designing effective and efficient ML data infrastructures.

6.8.4.1 Development Phase

In the development phase, storage systems play a critical role in supporting exploratory data analysis and iterative model development. This stage demands flexibility and collaboration, as data scientists often work with various datasets, experiment with feature engineering techniques, and rapidly iterate on model designs to refine their approaches.

One of the key challenges at this stage is managing the versions of datasets used in experiments. While traditional version control systems like Git excel at tracking code changes, they fall short when dealing with large datasets. This gap has led to the emergence of specialized tools like DVC (Data Version Control), which enable data scientists to efficiently track dataset changes, revert to previous versions, and share large files without duplication. These tools ensure that teams can maintain reproducibility and transparency throughout the iterative development process.

Balancing data accessibility and security further complicates the storage requirements in this phase. Data scientists require seamless access to datasets for experimentation, but organizations must simultaneously safeguard sensitive data. This tension often results in the implementation of sophisticated access control mechanisms, ensuring that datasets remain both accessible and protected. Secure data sharing systems enhance collaboration while adhering to strict organizational and regulatory requirements, enabling teams to work productively without compromising data integrity.

6.8.4.2 Training Phase

The training phase presents unique storage challenges due to the sheer volume of data processed and the computational intensity of model training. At this stage, the interplay between storage performance and computational efficiency becomes critical, as modern ML algorithms demand seamless integration between data access and processing.

To meet these demands, high-performance storage systems must provide the throughput required to feed data to multiple GPU or TPU accelerators simultaneously. Distributed training scenarios amplify this need, often requiring data transfer rates in the gigabytes per second range to ensure that accelerators remain fully utilized. This highlights the importance of optimizing storage for both capacity and speed.

Beyond data ingestion, managing intermediate results and checkpoints is another critical challenge in the training phase. Long-running training jobs frequently save intermediate model states to allow for resumption in case of interruptions. These checkpoints can grow significantly in size, especially for large-scale models, necessitating storage solutions that enable efficient saving and retrieval without impacting overall performance.

Complementing these systems is the concept of burst buffers[20], borrowed from high-performance computing. These high-speed, temporary storage layers are particularly valuable during training, as they can absorb large, bursty I/O operations. By buffering these spikes in demand, burst buffers help smooth out performance fluctuations and reduce the load on primary storage systems, ensuring that training pipelines remain efficient and reliable.

6.8.4.3 Deployment Phase

In the deployment and serving phase, the focus shifts from high-throughput batch operations during training to low-latency, often real-time, data access. This transition highlights the need to balance conflicting requirements, where storage systems must simultaneously support responsive model serving and enable continued learning in dynamic environments.

Real-time inference demands storage solutions capable of extremely fast access to model parameters and relevant features. To achieve this, systems often rely on in-memory databases or sophisticated caching strategies, ensuring that predictions can be made within milliseconds. These requirements become even more challenging in edge deployment scenarios, where devices operate with limited storage resources and intermittent connectivity to central data stores.

Adding to this complexity is the need to manage model updates in production environments. Storage systems must facilitate smooth transitions between model versions, ensuring minimal disruption to ongoing services. Techniques like shadow deployment, where new models run alongside existing ones for validation, allow organizations to iteratively roll out updates while monitoring their performance in real-world conditions.

6.8.4.4 Maintenance Phase

The monitoring and maintenance phase brings its own set of storage challenges, centered on ensuring the long-term reliability and performance of ML systems. At this stage, the focus shifts to capturing and analyzing data to monitor model behavior, detect issues, and maintain compliance with regulatory requirements.

[20] Burst Buffers: High-speed storage layers used to absorb large, temporary I/O demands in high-performance computing, smoothing performance during data-intensive operations.

A critical aspect of this phase is managing data drift, where the characteristics of incoming data change over time. Storage systems must efficiently capture and store incoming data along with prediction results, enabling ongoing analysis to detect and address shifts in data distributions. This ensures that models remain accurate and aligned with their intended use cases.

The sheer volume of logging and monitoring data generated by high-traffic ML services introduces questions of data retention and accessibility. Organizations must balance the need to retain historical data for analysis against the cost and complexity of storing it. Strategies such as tiered storage and compression can help manage costs while ensuring that critical data remains accessible when needed.

Regulated industries often require immutable storage to support auditing and compliance efforts. Storage systems designed for this purpose guarantee data integrity and non-repudiability, ensuring that stored data cannot be altered or deleted. Blockchain-inspired solutions and write-once-read-many (WORM) technologies are commonly employed to meet these stringent requirements.

6.8.5 Feature Storage

Feature stores are a centralized repository that stores and serves pre-computed features for machine learning models, ensuring consistency between training and inference workflows. They have emerged as a critical component in the ML infrastructure stack, addressing the unique challenges of managing and serving features for machine learning models. They act as a central repository for storing, managing, and serving machine learning features, bridging the gap between data engineering and machine learning operations.

What makes feature stores particularly interesting is their role in solving several key challenges in ML pipelines. First, they address the problem of feature consistency between training and serving environments. In traditional ML workflows, features are often computed differently in offline (training) and online (serving) environments, leading to discrepancies that can degrade model performance. Feature stores provide a single source of truth for feature definitions, ensuring consistency across all stages of the ML lifecycle.

Another fascinating aspect of feature stores is their ability to promote feature reuse across different models and teams within an organization. By centralizing feature computation and storage, feature stores can significantly reduce redundant work. For instance, if multiple teams are working on different models that require similar features (e.g., customer lifetime value in a retail context), these features can be computed once and reused across projects, improving efficiency and consistency.

Feature stores also play a role in managing the temporal aspects of features. Many ML use cases require correct point-in-time feature values, especially in scenarios involving time-series data or where historical context is important. Feature stores typically offer time-travel capabilities, allowing data scientists to retrieve feature values as they were at any point in the past. This is crucial for training models on historical data and for ensuring consistency between training and serving environments, as illustrated in Figure 6.16 which shows how data flows through these systems to eventually yield a model.

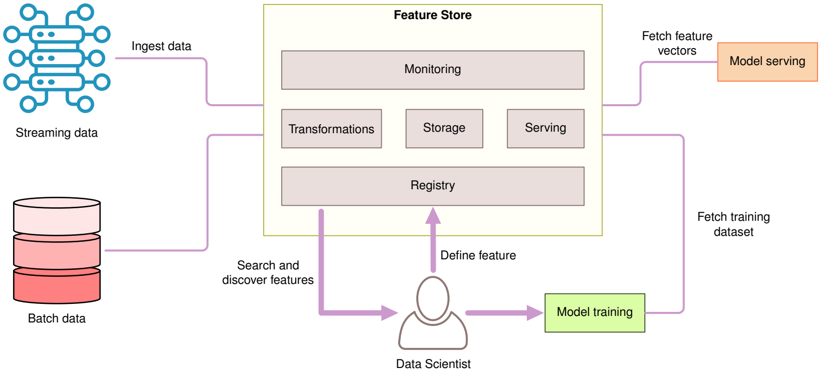


Figure 6.16: High-level overview of feature stores interact with data, users and model training and deployment.

The performance characteristics of feature stores are particularly intriguing from a storage perspective. They need to support both high-throughput batch retrieval for model training and low-latency lookups for online inference. This often leads to hybrid architectures where feature stores maintain both an offline store (optimized for batch operations) and an online store (optimized for real-time serving). Synchronization between these stores becomes a critical consideration.

Feature stores also introduce interesting challenges in terms of data freshness and update strategies. Some features may need to be updated in real-time (e.g., current user session information), while others might be updated on a daily or weekly basis (e.g., aggregated customer behavior metrics). Managing these different update frequencies and ensuring that the most up-to-date features are always available for inference can be complex.

From a storage perspective, feature stores often leverage a combination of different storage technologies to meet their diverse requirements. This might include columnar storage formats like Parquet for the offline store, in-memory databases or key-value stores for the online store, and streaming platforms like Apache Kafka for real-time feature updates.

6.8.6 Caching Techniques

Caching plays a role in optimizing the performance of ML systems, particularly in scenarios involving frequent data access or computation-intensive operations. In the context of machine learning, caching strategies extend beyond traditional web or database caching, addressing unique challenges posed by ML workflows.

One of the primary applications of caching in ML systems is in feature computation and serving. Many features used in ML models are computationally expensive to calculate, especially those involving complex aggregations or time-window operations. By caching these computed features, systems can significantly reduce latency in both training and inference scenarios. For instance, in a recommendation system, caching user embedding vectors can dramatically speed up the generation of personalized recommendations.

Caching strategies in ML systems often need to balance between memory usage and computation time. This trade-off is particularly evident in large-scale distributed training scenarios. Caching frequently accessed data shards or mini-batches in memory can significantly reduce I/O overhead, but it requires careful memory management to avoid out-of-memory errors, especially when working with large datasets or models.

Another interesting application of caching in ML systems is model caching. In scenarios where multiple versions of a model are deployed (e.g., for A/B testing or gradual rollout), caching the most frequently used model versions in memory can significantly reduce inference latency. This becomes especially important in edge computing scenarios, where storage and computation resources are limited.

Caching also plays a vital role in managing intermediate results in ML pipelines. For instance, in feature engineering pipelines that involve multiple transformation steps, caching intermediate results can prevent redundant computations when rerunning pipelines with minor changes. This is particularly useful during the iterative process of model development and experimentation.

One of the challenges in implementing effective caching strategies for ML is managing cache invalidation and updates. ML systems often deal with dynamic data where feature values or model parameters may change over time. Implementing efficient cache update mechanisms that balance between data freshness and system performance is an ongoing area of research and development.

Distributed caching becomes particularly important in large-scale ML systems. Technologies like Redis or Memcached are often employed to create distributed caching layers that can serve multiple training or inference nodes. These distributed caches need to handle challenges like maintaining consistency across nodes and managing failover scenarios.

Edge caching is another fascinating area in ML systems, especially with the growing trend of edge AI. In these scenarios, caching strategies need to account for limited storage and computational resources on edge devices, as well as potentially intermittent network connectivity. Intelligent caching strategies that prioritize the most relevant data or model components for each edge device can significantly improve the performance and reliability of edge ML systems.

Lastly, the concept of semantic caching[21] is gaining traction in ML systems. Unlike traditional caching that operates on exact matches, semantic caching attempts to reuse cached results for semantically similar queries. This can be particularly useful in ML systems where slight variations in input may not significantly change the output, potentially leading to substantial performance improvements.

6.8.7 Data Access Patterns

Understanding the access patterns in ML systems is useful for designing efficient storage solutions and optimizing the overall system performance. ML workloads exhibit distinct data access patterns that often differ significantly from traditional database or analytics workloads.

One of the most prominent access patterns in ML systems is sequential reading of large datasets during model training. Unlike transactional systems that typically access small amounts of data randomly, ML training often involves

[21] Semantic Caching: A caching technique that reuses results of previous computations for semantically similar queries, reducing redundancy in data processing.

reading entire datasets multiple times (epochs) in a sequential manner. This pattern is particularly evident in deep learning tasks, where large volumes of data are fed through neural networks repeatedly. Storage systems optimized for high-throughput sequential reads, such as distributed file systems or object stores, are well-suited for this access pattern.

However, the sequential read pattern is often combined with random shuffling between epochs to prevent overfitting and improve model generalization. This introduces an interesting challenge for storage systems, as they need to efficiently support both sequential and random access patterns, often within the same training job.

In contrast to the bulk sequential reads common in training, inference workloads often require fast random access to specific data points or features. For example, a recommendation system might need to quickly retrieve user and item features for real-time personalization. This necessitates storage solutions with low-latency random read capabilities, often leading to the use of in-memory databases or caching layers.

Feature stores, which we discussed earlier, introduce their own unique access patterns. They typically need to support both high-throughput batch reads for offline training and low-latency point lookups for online inference. This dual-nature access pattern often leads to the implementation of separate offline and online storage layers, each optimized for its specific access pattern.

Time-series data, common in many ML applications such as financial forecasting or IoT analytics, presents another interesting access pattern. These workloads often involve reading contiguous blocks of time-ordered data, but may also require efficient retrieval of specific time ranges or periodic patterns. Specialized time-series databases or carefully designed partitioning schemes in general-purpose databases are often employed to optimize these access patterns.

Another important consideration is the write access pattern in ML systems. While training workloads are often read-heavy, there are scenarios that involve significant write operations. For instance, continual learning systems may frequently update model parameters, and online learning systems may need to efficiently append new training examples to existing datasets.

Understanding these diverse access patterns is helpful in designing and optimizing storage systems for ML workloads. It often leads to hybrid storage architectures that combine different technologies to address various access patterns efficiently. For example, a system might use object storage for large-scale sequential reads during training, in-memory databases for low-latency random access during inference, and specialized time-series storage for temporal data analysis.

As ML systems continue to evolve, new access patterns are likely to emerge, driving further innovation in storage technologies and architectures. The challenge lies in creating flexible, scalable storage solutions that can efficiently support the diverse and often unpredictable access patterns of modern ML workloads.

6.8.8 Continuing the KWS Example

During development and training, KWS systems must efficiently store and manage large collections of audio data. This includes raw audio recordings from various sources (crowd-sourced, synthetic, and real-world captures), processed features (like spectrograms or MFCCs), and model checkpoints. A typical architecture might use a data lake for raw audio files, allowing flexible storage of diverse audio formats, while processed features are stored in a more structured data warehouse for efficient access during training.

KWS systems benefit significantly from feature stores, particularly for managing pre-computed audio features. For example, commonly used spectrogram representations or audio embeddings can be computed once and stored for reuse across different experiments or model versions. The feature store must handle both batch access for training and real-time access for inference, often implementing a dual storage architecture, which includes an offline store for training data and an online store for low-latency inference.

In production, KWS systems require careful consideration of edge storage requirements. The models must be compact enough to fit on resource-constrained devices while maintaining quick access to necessary parameters for real-time wake word detection. This often involves optimized storage formats and careful caching strategies to balance between memory usage and inference speed.

6.9 Data Governance

Data governance is a significant component in the development and deployment of ML systems. It encompasses a set of practices and policies that ensure data is accurate, secure, compliant, and ethically used throughout the ML lifecycle. As ML systems become increasingly integral to decision-making processes across various domains, the importance of robust data governance has grown significantly.

One of the central challenges of data governance is addressing the unique complexities posed by ML workflows. These workflows often involve opaque processes, such as feature engineering and model training, which can obscure how data is being used. As shown in Figure 6.17, governance practices aim to tackle these issues by focusing on maintaining data privacy, ensuring fairness, and providing transparency in decision-making processes. These practices go beyond traditional data management to address the evolving needs of ML systems.

Security and access control form an essential aspect of data governance. Implementing measures to protect data from unauthorized access or breaches is critical in ML systems, which often deal with sensitive or proprietary information. For instance, a healthcare application may require granular access controls to ensure that only authorized personnel can view patient data. Encrypting data both at rest and in transit is another common approach to safeguarding information while enabling secure collaboration among ML teams.

Privacy protection is another key pillar of data governance. As ML models often rely on large-scale datasets, there is a risk of infringing on individual privacy rights. Techniques such as differential privacy[22] can address this concern

[22] Differential Privacy: A technique that adds randomness to dataset queries to protect individual data privacy while maintaining overall data utility.

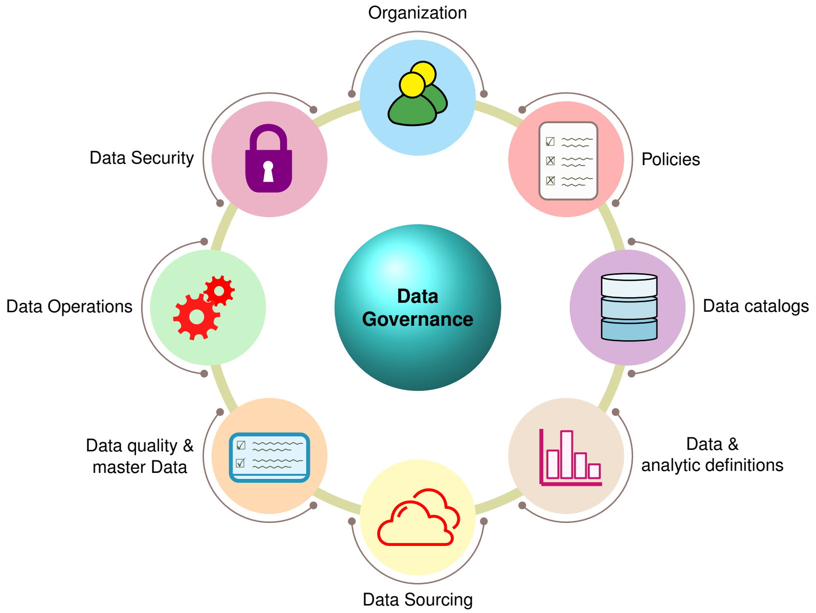
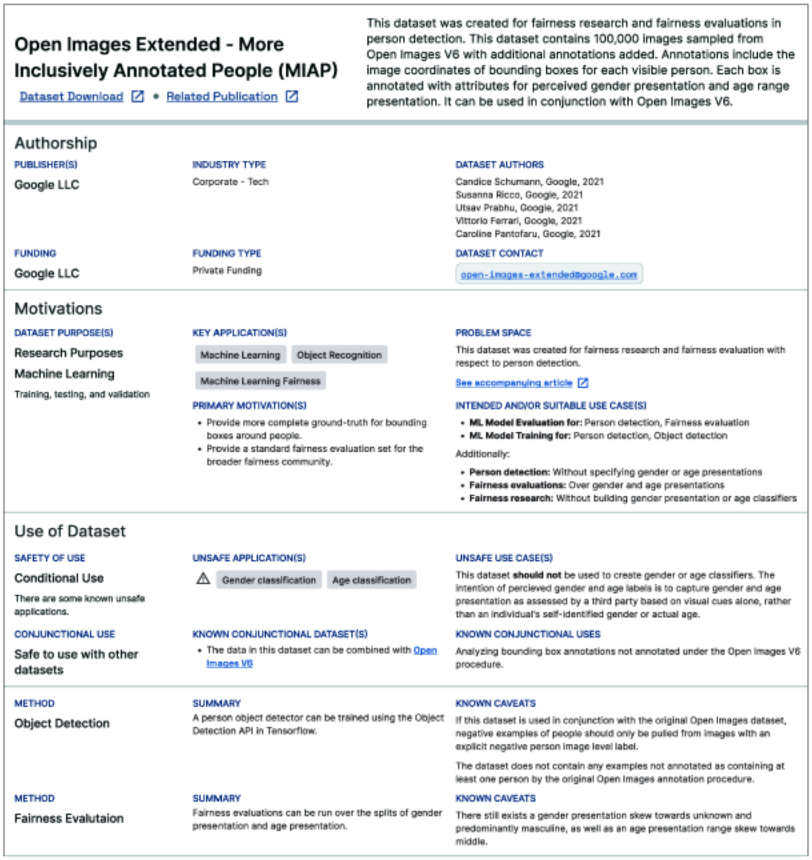


Figure 6.17: The many pillars of data governance.

by adding carefully calibrated noise to the data. This ensures that individual identities are protected while preserving the statistical patterns necessary for model training. These techniques allow ML systems to benefit from data-driven insights without compromising ethical considerations (Dwork, n.d.), which we will learn more about in the Responsible AI chapter.

Regulatory compliance is a critical area where data governance plays a central role. Laws such as the GDPR in Europe and the HIPAA in the United States impose strict requirements on data handling. Compliance with these regulations often involves implementing features like the ability to delete data upon request or providing individuals with copies of their data, and a "right to explanation" on decisions made by algorithms (Wachter, Mittelstadt, and Russell 2017). These measures not only protect individuals but also ensure organizations avoid legal and reputational risks.

Documentation and metadata management, which are often less discussed, are just as important for transparency and reproducibility in ML systems. Clear records of data lineage, including how data flows and transforms throughout the ML pipeline, are essential for accountability. Standardized documentation frameworks, such as Data Cards proposed by Pushkarna, Zaldivar, and Kjartansson (2022), offer a structured way to document the characteristics, limitations, and potential biases of datasets. For example, as shown in Figure 6.18, the Open Images Extended, More Inclusively Annotated People (MIAP) dataset, uses a data card to provide detailed information about its motivations, intended use cases, and known risks. This type of documentation enables developers to evaluate datasets effectively and promotes responsible use.



Open Images Extended - More Inclusively Annotated People (MIAP)

Dataset Download • Related Publication

This dataset was created for fairness research and fairness evaluations in person detection. This dataset contains 100,000 images sampled from Open Images V6 with additional annotations added. Annotations include the image coordinates of bounding boxes for each visible person. Each box is annotated with attributes for perceived gender presentation and age range presentation. It can be used in conjunction with Open Images V6.

Authorship

PUBLISHER(S)
Google LLC

INDUSTRY TYPE
Corporate - Tech

DATASET AUTHORS
Candice Schumann, Google, 2021
Susanna Ricco, Google, 2021
Utsav Prabhu, Google, 2021
Vittorio Ferrari, Google, 2021
Caroline Pantofaru, Google, 2021

FUNDING
Google LLC

FUNDING TYPE
Private Funding

DATASET CONTACT
open-images-extended@google.com

Motivations

DATASET PURPOSE(S)
Research Purposes
Machine Learning
Training, testing, and validation

KEY APPLICATION(S)
Machine Learning · Object Recognition · Machine Learning Fairness

PRIMARY MOTIVATION(S)

- Provide more complete ground-truth for bounding boxes around people.
- Provide a standard fairness evaluation set for the broader fairness community.

PROBLEM SPACE
This dataset was created for fairness research and fairness evaluation with respect to person detection.
See accompanying article

INTENDED AND/OR SUITABLE USE CASE(S)

- **ML Model Evaluation for:** Person detection, Fairness evaluation
- **ML Model Training for:** Person detection, Object detection

Additionally:

- **Person detection:** Without specifying gender or age presentations
- **Fairness evaluations:** Over gender and age presentations
- **Fairness research:** Without building gender presentation or age classifiers

Use of Dataset

SAFETY OF USE
Conditional Use
There are some known unsafe applications.

UNSAFE APPLICATION(S)
⚠ Gender classification · Age classification

UNSAFE USE CASE(S)
This dataset **should not** be used to create gender or age classifiers. The intention of percieved gender and age labels is to capture gender and age presentation as assessed by a third party based on visual cues alone, rather than an individual's self-identified gender or actual age.

CONJUNCTIONAL USE
Safe to use with other datasets

KNOWN CONJUNCTIONAL DATASET(S)

- The data in this dataset can be combined with Open Images V6

KNOWN CONJUNCTIONAL USES
Analyzing bounding box annotations not annotated under the Open Images V6 procedure.

METHOD
Object Detection

SUMMARY
A person object detector can be trained using the Object Detection API in Tensorflow.

KNOWN CAVEATS
If this dataset is used in conjunction with the original Open Images dataset, negative examples of people should only be pulled from images with an explicit negative person image level label.

The dataset does not contain any examples not annotated as containing at least one person by the original Open Images annotation procedure.

METHOD
Fairness Evalutaion

SUMMARY
Fairness evaluations can be run over the splits of gender presentation and age presentation.

KNOWN CAVEATS
There still exists a gender presentation skew towards unknown and predominantly masculine, as well as an age presentation range skew towards middle.

Figure 6.18: Data card example for the Open Images Extended dataset.

Audit trails are another important component of data governance. These detailed logs track data access and usage throughout the lifecycle of ML models, from collection to deployment. Comprehensive audit trails are invaluable for troubleshooting and accountability, especially in cases of data breaches or unexpected model behavior. They help organizations understand what actions were taken and why, providing a clear path for resolving issues and ensuring compliance.

Consider a hypothetical ML system designed to predict patient outcomes in a hospital. Such a system would need to address several governance challenges. It would need to ensure that patient data is securely stored and accessed only by authorized personnel, with privacy-preserving techniques in place to protect individual identities. The system would also need to comply with healthcare regulations governing the use of patient data, including detailed documentation of how data is processed and transformed. Comprehensive audit logs would be necessary to track data usage and ensure accountability.

As ML systems grow more complex and influential, the challenges of data governance will continue to evolve. Emerging trends, such as blockchain-inspired technologies for tamper-evident logs and automated governance tools, offer promising solutions for real-time monitoring and issue detection. By adopting robust data governance practices, including tools like Data Cards, organizations can build ML systems that are transparent, ethical, and trustworthy.

6.10 Conclusion

Data engineering is the backbone of any successful ML system. By thoughtfully defining problems, designing robust pipelines, and practicing rigorous data governance, teams establish a foundation that directly influences model performance, reliability, and ethical standing. Effective data acquisition strategies, whether by utilizing existing datasets, employing web scraping techniques, or engaging in crowdsourcing, must balance the realities of domain constraints, privacy obligations, and labeling complexities. Likewise, decisions around data ingestion (batch or streaming) and transformation (ETL or ELT) affect both cost and throughput, with monitoring and observability essential to detect shifting data quality.

Throughout this chapter, we saw how critical it is to prepare data well in advance of modeling. Data labeling emerges as a particularly delicate phase: it involves human effort, requires strong quality control practices, and has ethical ramifications. Storage choices, such as relational databases, data warehouses, data lakes, or specialized systems, must align with both the volume and velocity of ML workloads. Feature stores and caching strategies support efficient retrieval across training and serving pipelines, while good data governance ensures adherence to legal regulations, protects privacy, and maintains stakeholder trust.

All these elements interlock to create an ecosystem that reliably supplies ML models with the high-quality data they need. When done well, data engineering empowers teams to iterate faster, confidently deploy new features, and build systems capable of adapting to real-world complexity. The next chapters will build on these foundations, exploring how optimized training, robust model operations, and security considerations together form a holistic approach to delivering AI solutions that perform reliably and responsibly at scale.

6.11 Resources

ℹ Slides

- Data Engineering: Overview.
- Feature engineering.
- Data Standards: Speech Commands.
- Crowdsourcing Data for the Long Tail.
- Reusing and Adapting Existing Datasets.
- Responsible Data Collection.
- Data Anomaly Detection:

  - Anomaly Detection: Overview.
  - Anomaly Detection: Challenges.
  - Anomaly Detection: Datasets.
  - Anomaly Detection: using Autoencoders.

! Videos

- *Coming soon.*

Exercises

- *Coming soon.*

Chapter 7

AI Frameworks

Figure 7.1: *DALL·E 3 Prompt: Illustration in a rectangular format, designed for a professional textbook, where the content spans the entire width. The vibrant chart represents training and inference frameworks for ML. Icons for TensorFlow, Keras, PyTorch, ONNX, and TensorRT are spread out, filling the entire horizontal space, and aligned vertically. Each icon is accompanied by brief annotations detailing their features. The lively colors like blues, greens, and oranges highlight the icons and sections against a soft gradient background. The distinction between training and inference frameworks is accentuated through color-coded sections, with clean lines and modern typography maintaining clarity and focus.*

Purpose

How do AI frameworks bridge the gap between theoretical design and practical implementation, and what role do they play in enabling scalable and effiicent machine learning systems?

AI frameworks are the middleware software layer that transforms abstract model specifications into executable implementations. The evolution of these frameworks reveals fundamental patterns for translating high-level designs into efficient computational workflows and system execution. Their architecture shines light on the essential trade-offs between abstraction, performance, and portability, providing systematic approaches to managing complexity in machine learning systems. Understanding framework capabilities and constraints offers insights into the engineering decisions that shape system scalability, enabling the development of robust, deployable solutions across diverse computing environments.

Learning Objectives

- Trace the evolution of machine learning frameworks from early numerical libraries to modern deep learning systems
- Analyze framework fundamentals including tensor data structures, computational graphs, execution models, and memory management
- Differentiate between machine learning frameworks architectures, execution strategies, and development tools
- Compare framework specializations across cloud, edge, mobile, and TinyML applications

7.1 Overview

Modern machine learning development relies fundamentally on machine learning frameworks, which are comprehensive software libraries or platforms designed to simplify the development, training, and deployment of machine learning models. These frameworks play multiple roles in ML systems, much like operating systems are the foundation of computing systems. Just as operating systems abstract away the complexity of hardware resources and provide standardized interfaces for applications, ML frameworks abstract the intricacies of mathematical operations and hardware acceleration, providing standardized APIs for ML development.

The capabilities of ML frameworks are diverse and continuously evolving. They provide efficient implementations of mathematical operations, automatic differentiation capabilities, and tools for managing model development, hardware acceleration, and memory utilization. For production systems, they offer standardized approaches to model deployment, versioning, and optimization. However, due to their diversity, there is no universally agreed-upon definition of an ML framework. To establish clarity for this chapter, we adopt the following definition:

Framework Definition

A **Machine Learning Framework (ML Framework)** is a *software platform* that provides tools and abstractions for designing, training, and deploying machine learning models. It bridges *user applications* with *infrastructure*, enabling *algorithmic expressiveness* through computational graphs and operators, *workflow orchestration* across the machine learning lifecycle, *hardware optimization* with schedulers and compilers, *scalability* for distributed and edge systems, and *extensibility* to support diverse use cases. ML frameworks form the foundation of modern machine learning systems by simplifying development and deployment processes.

The landscape of ML frameworks continues to evolve with the field itself. Today's frameworks must address diverse requirements: from training large language models on distributed systems to deploying compact neural networks on tiny IoT devices. Popular frameworks like PyTorch and TensorFlow[0] have developed rich ecosystems that extend far beyond basic model implementation, encompassing tools for data preprocessing, model optimization, and deployment.

As we progress into examining training, optimization, and deployment, understanding ML frameworks becomes necessary as they orchestrate the entire machine learning lifecycle. These frameworks provide the architecture that connects all aspects of ML systems, from data ingestion to model deployment. Just as understanding a blueprint is important before studying construction techniques, grasping framework architecture is vital before diving into training methodologies and deployment strategies. Modern frameworks encapsulate the complete ML workflow, and their design choices influence how we approach training, optimization, and inference.

This chapter helps us learn how these complex frameworks function, their architectural principles, and their role in modern ML systems. Understanding these concepts will provide the necessary context as we explore specific aspects of the ML lifecycle in subsequent chapters.

7.2 Evolution History

The evolution of machine learning frameworks mirrors the broader development of artificial intelligence and computational capabilities. This section explores the distinct phases that reflect both technological advances and changing requirements of the AI community, from early numerical computing libraries to modern deep learning frameworks.

7.2.1 Evolution Timeline

The development of machine learning frameworks has been built upon decades of foundational work in computational libraries. From the early building blocks of BLAS and LAPACK to today's cutting-edge frameworks like TensorFlow, PyTorch, and JAX, this journey represents a steady progression toward higher-level abstractions that make machine learning more accessible and powerful.

Looking at Figure 7.2, we can trace how these fundamental numerical computing libraries laid the groundwork for modern ML development. The mathematical foundations established by BLAS and LAPACK enabled the creation of more user-friendly tools like NumPy and SciPy, which in turn set the stage for today's sophisticated deep learning frameworks.

This evolution reflects a clear trend: each new layer of abstraction has made complex computational tasks more approachable while building upon the robust foundations of its predecessors. Let us examine how these systems built on top of one another.

7.2.2 Early Numerical Libraries

The foundation for modern ML frameworks begins at the most fundamental level of computation: matrix operations. Machine learning computations are primarily matrix-matrix and matrix-vector multiplications. The Basic Linear

[0] TensorFlow and PyTorch: TensorFlow, developed by Google, excels in production deployment and offers TensorFlow Lite for mobile/embedded applications. PyTorch, developed by Meta AI, is widely adopted in research settings due to its dynamic computation model and developer-friendly features. Together they represent the two most prevalent deep learning frameworks.

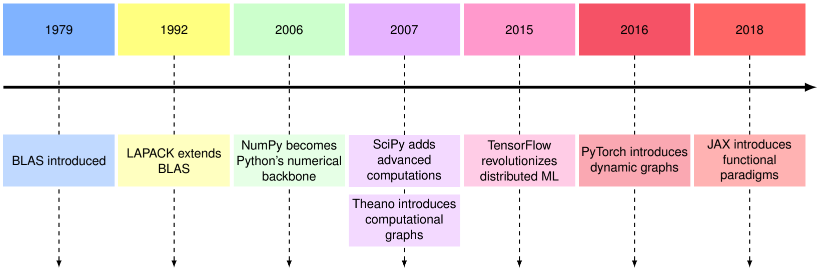

1979
1992
2006
2007
2015
2016
2018
BLAS introduced
LAPACK extends BLAS
NumPy becomes Python's numerical backbone
SciPy adds advanced computations
Theano introduces computational graphs
TensorFlow revolutionizes distributed ML
PyTorch introduces dynamic graphs
JAX introduces functional paradigms

Figure 7.2: Timeline of major developments in computational libraries and machine learning frameworks.

Algebra Subprograms (BLAS), developed in 1979, provided these essential matrix operations that would become the computational backbone of machine learning (H. T. Kung and Leiserson 1979). These low-level operations, when combined and executed efficiently, enable the complex calculations required for training neural networks and other ML models.

Building upon BLAS, the Linear Algebra Package (LAPACK) emerged in 1992, extending these capabilities with more sophisticated linear algebra operations such as matrix decompositions, eigenvalue problems, and linear system solutions. This layered approach of building increasingly complex operations from fundamental matrix computations became a defining characteristic of ML frameworks.

The development of NumPy in 2006 marked an important milestone in this evolution, building upon its predecessors Numeric and Numarray to become the fundamental package for numerical computation in Python. NumPy introduced n-dimensional array objects and essential mathematical functions, but more importantly, it provided an efficient interface to these underlying BLAS and LAPACK operations. This abstraction allowed developers to work with high-level array operations while maintaining the performance of optimized low-level matrix computations.

In 2001, SciPy emerged as a powerful extension built on top of NumPy, adding specialized functions for optimization, linear algebra, and signal processing. This further exemplified the pattern of progressive abstraction in ML frameworks: from basic matrix operations to sophisticated numerical computations, and eventually to high-level machine learning algorithms. This layered architecture, starting from fundamental matrix operations and building upward, would become a blueprint for future ML frameworks, as we will see in this chapter.

7.2.3 First-Generation Frameworks

The transition from numerical libraries to dedicated machine learning frameworks marked an important evolution in abstraction. While the underlying computations remained rooted in matrix operations, frameworks began to encapsulate these operations into higher-level machine learning primitives. The University of Waikato introduced Weka in 1993 (Witten and Frank 2002), one of the earliest ML frameworks, which abstracted matrix operations into data mining tasks, though it was limited by its Java implementation and focus on smaller-scale computations.

Scikit-learn, emerging in 2007, was a significant advancement in this abstraction. Building upon the NumPy and SciPy foundation, it transformed basic matrix operations into intuitive ML algorithms. For example, what was fundamentally a series of matrix multiplications and gradient computations became a simple `fit()` method call in a logistic regression model. This abstraction pattern - hiding complex matrix operations behind clean APIs - would become a defining characteristic of modern ML frameworks.

Theano, which appeared in 2007, was a major advancement, which was developed at the Montreal Institute for Learning Algorithms, MILA, Theano introduced two revolutionary concepts: computational graphs and GPU acceleration (Team et al. 2016). Computational graphs represented mathematical operations as directed graphs, with matrix operations as nodes and data flowing between them. This graph-based approach allowed for automatic differentiation and optimization of the underlying matrix operations. More importantly, it enabled the framework to automatically route these operations to GPU hardware, dramatically accelerating matrix computations.

Meanwhile, Torch, created at NYU in 2002, took a different approach to handling matrix operations. It emphasized immediate execution of operations (eager execution) and provided a flexible interface for neural network implementations. Torch's design philosophy of prioritizing developer experience while maintaining high performance influenced many subsequent frameworks. Its architecture demonstrated how to balance high-level abstractions with efficient low-level matrix operations, establishing design patterns that would later influence frameworks like PyTorch.

7.2.4 Emergence of Deep Learning Frameworks

The deep learning revolution demanded a fundamental shift in how frameworks handled matrix operations, primarily due to three factors: the massive scale of computations, the complexity of gradient calculations through deep networks, and the need for distributed processing. Traditional frameworks, designed for classical machine learning algorithms, could not efficiently handle the billions of matrix operations required for training deep neural networks.

The foundations for modern deep learning frameworks emerged from academic research. The University of Montreal's Theano, released in 2007, established the concepts that would shape future frameworks (Bergstra et al. 2010). It introduced key concepts such as computational graphs[1] for automatic differentiation and GPU acceleration, which we will explore in more detail later in this chapter, demonstrating how to efficiently organize and optimize complex neural network computations.

Caffe, released by UC Berkeley in 2013, advanced this evolution by introducing specialized implementations of convolutional operations (Y. Jia et al. 2014). While convolutions are mathematically equivalent to specific patterns of matrix multiplication, Caffe optimized these patterns specifically for computer vision tasks, demonstrating how specialized matrix operation implementations could dramatically improve performance for specific network architectures.

Google's TensorFlow, introduced in 2015, revolutionized the field by treating matrix operations as part of a distributed computing problem (Jeffrey Dean

[1] Computational Graph: A representation of mathematical computations as a directed graph, where nodes represent operations and edges represent data dependencies, used to enable automatic differentiation.

and Ghemawat 2008). It represented all computations, from individual matrix multiplications to entire neural networks, as a static computational graph that could be split across multiple devices. This approach enabled training of unprecedented model sizes by distributing matrix operations across clusters of computers and specialized hardware. TensorFlow's static graph approach, while initially constraining, allowed for aggressive optimization of matrix operations through techniques like kernel fusion (combining multiple operations into a single kernel for efficiency) and memory planning (pre-allocating memory for operations).

Microsoft's CNTK entered the landscape in 2016, bringing robust implementations for speech recognition and natural language processing tasks (Seide and Agarwal 2016). Its architecture emphasized scalability across distributed systems while maintaining efficient computation for sequence-based models.

Facebook's PyTorch, also launched in 2016, took a radically different approach to handling matrix computations. Instead of static graphs, PyTorch introduced dynamic computational graphs that could be modified on the fly (Paszke et al. 2019). This dynamic approach, while potentially sacrificing some optimization opportunities, made it much easier for researchers to debug and understand the flow of matrix operations in their models. PyTorch's success demonstrated that the ability to introspect and modify computations dynamically was as important as raw performance for many applications.

Amazon's MXNet approached the challenge of large-scale matrix operations by focusing on memory efficiency and scalability across different hardware configurations. It introduced a hybrid approach that combined aspects of both static and dynamic graphs, allowing for flexible model development while still enabling aggressive optimization of the underlying matrix operations.

As deep learning applications grew more diverse, the need for specialized and higher-level abstractions became apparent. Keras emerged in 2015 to address this need, providing a unified interface that could run on top of multiple lower-level frameworks (Chollet et al. 2015).

Google's JAX, introduced in 2018, brought functional programming principles to deep learning computations, enabling new patterns of model development (Bradbury et al. 2018). FastAI built upon PyTorch to package common deep learning patterns into reusable components, making advanced techniques more accessible to practitioners (J. Howard and Gugger 2020). These higher-level frameworks demonstrated how abstraction could simplify development while maintaining the performance benefits of their underlying implementations.

7.2.5 Hardware Impact on Design

Hardware developments have fundamentally reshaped how frameworks implement and optimize matrix operations. The introduction of NVIDIA's CUDA platform in 2007 marked a pivotal moment in framework design by enabling general-purpose computing on GPUs.[2] This was transformative because GPUs excel at parallel matrix operations, offering orders of magnitude speedup for the computations in deep learning. While a CPU might process matrix ele-

[2] GPUs are designed for rendering graphics and is heavily used for parallel processing. TPUs were developed by Google for fast matrix multiplication and deep learning tasks.

ments sequentially, a GPU can process thousands of elements simultaneously, fundamentally changing how frameworks approach computation scheduling.

The development of hardware-specific accelerators further revolutionized framework design. Google's Tensor Processing Units (TPUs), first deployed in 2016, were purpose-built for tensor operations, the fundamental building blocks of deep learning computations. TPUs introduced systolic array architectures[3], which are particularly efficient for matrix multiplication and convolution operations. This hardware architecture prompted frameworks like TensorFlow to develop specialized compilation strategies that could map high-level operations directly to TPU instructions, bypassing traditional CPU-oriented optimizations.

Mobile hardware accelerators,[4] such as Apple's Neural Engine (2017) and Qualcomm's Neural Processing Units, brought new constraints and opportunities to framework design. These devices emphasized power efficiency over raw computational speed, requiring frameworks to develop new strategies for quantization and operator fusion[5]. Mobile frameworks like TensorFlow Lite (more recently rebraneded to LiteRT) and PyTorch Mobile needed to balance model accuracy with energy consumption, leading to innovations in how matrix operations are scheduled and executed.

The emergence of custom ASIC (Application-Specific Integrated Circuit)[6] solutions has further diversified the hardware landscape. Companies like Graphcore, Cerebras, and SambaNova have developed unique architectures for matrix computation, each with different strengths and optimization opportunities. This proliferation of specialized hardware has pushed frameworks to adopt more flexible intermediate representations of matrix operations, allowing for target-specific optimization while maintaining a common high-level interface.

Field Programmable Gate Arrays (FPGAs) introduced yet another dimension to framework optimization. Unlike fixed-function ASICs, FPGAs allow for reconfigurable circuits that can be optimized for specific matrix operation patterns. Frameworks responding to this capability developed just-in-time compilation strategies that could generate optimized hardware configurations based on the specific needs of a model.

7.3 Fundamental Concepts

Modern machine learning frameworks operate through the integration of four key layers: Fundamentals, Data Handling, Developer Interface, and Execution and Abstraction. These layers function together to provide a structured and efficient foundation for model development and deployment, as illustrated in Figure 7.3.

The Fundamentals layer establishes the structural basis of these frameworks through computational graphs. These graphs represent the operations within a model as directed acyclic graphs (DAGs), enabling automatic differentiation and optimization. By organizing operations and data dependencies, computational graphs provide the framework with the ability to distribute workloads and execute computations efficiently across a variety of hardware platforms.

The Data Handling layer manages numerical data and parameters essential for machine learning workflows. Central to this layer are specialized data structures, such as tensors, which handle high-dimensional arrays while optimizing

[3] Systolic Array: A hardware architecture designed to perform a series of parallel computations in a time-synchronized manner, optimizing the flow of data through a grid of processors for tasks like matrix multiplication.

[4] Hardware accelerators are specialized systems that perform computing tasks more efficiently than central processing units (CPUs). These accelerators speed up the computation by allowing greater concurrency, optimized matrix operations, simpler control logic, and dedicated memory architecture. Each processing unit is more specialized than a CPU core, so more units can be fit on a chip and run in unison.

[5] Operation fusion: A technique that combines multiple consecutive operations into a single kernel to reduce memory bandwidth usage and improve computational efficiency, particularly for element-wise operations.

[6] Application-Specific Integrated Circuit (ASIC): is a custom-built hardware chip optimized for specific tasks, such as matrix computations in deep learning, offering superior performance and energy efficiency compared to general-purpose processors.

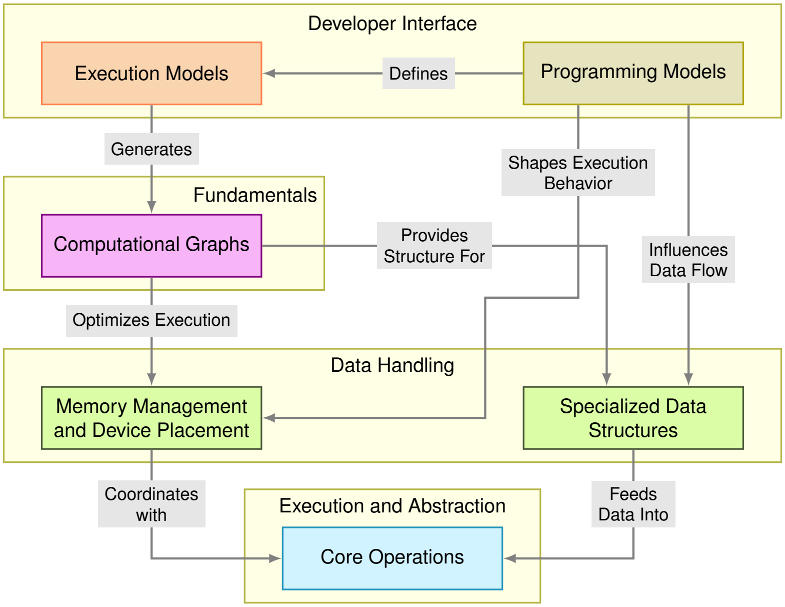


Figure 7.3: Framework component interaction.

memory usage and device placement. Additionally, memory management and data movement strategies ensure that computational workloads are executed efficiently, particularly in environments with diverse or limited hardware resources.

The Developer Interface layer provides the tools and abstractions through which users interact with the framework. Programming models allow developers to define machine learning algorithms in a manner suited to their specific needs. These are categorized as either imperative or symbolic. Imperative models offer flexibility and ease of debugging, while symbolic models prioritize performance and deployment efficiency. Execution models further shape this interaction by defining whether computations are carried out eagerly (immediately) or as pre-optimized static graphs.

The Execution and Abstraction layer transforms these high-level representations into efficient hardware-executable operations. Core operations, encompassing everything from basic linear algebra to complex neural network layers, are highly optimized for diverse hardware platforms. This layer also includes mechanisms for allocating resources and managing memory dynamically, ensuring robust and scalable performance in both training and inference settings.

Understanding these interconnected layers is essential for leveraging machine learning frameworks effectively. Each layer plays a distinct yet interdependent role in facilitating experimentation, optimization, and deployment. By mastering these concepts, practitioners can make informed decisions about resource utilization, scaling strategies, and the suitability of specific frameworks for various tasks.

7.3.1 Computational Graphs

Machine learning frameworks must efficiently translate high-level model descriptions into executable computations across diverse hardware platforms. At the center of this translation lies the computational graph, a powerful abstraction that represents mathematical operations and their dependencies. We begin by examining the fundamental structure of computational graphs, then investigate their implementation in modern frameworks, and analyze their implications for system design and performance.

7.3.1.1 Basic Concepts

Computational graphs emerged as a fundamental abstraction in machine learning frameworks to address the growing complexity of deep learning models. As models grew larger and more sophisticated, the need for efficient execution across diverse hardware platforms became crucial. The computational graph bridges the gap between high-level model descriptions and low-level hardware execution (Baydin et al. 2017a), representing a machine learning model as a directed acyclic graph (DAG) where nodes represent operations and edges represent data flow.

For example, a node might represent a matrix multiplication operation, taking two input matrices (or tensors) and producing an output matrix (or tensor). To visualize this, consider the simple example in Figure 7.4. The directed acyclic graph computes $z = x \times y$, where each variable is just numbers.

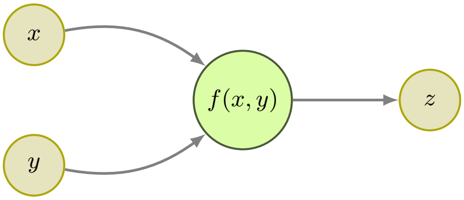


Figure 7.4: Basic example of a computational graph.

As shown in Figure 7.5, the structure of the computation graph[7] involves defining interconnected layers, such as convolution, activation, pooling, and normalization, which are optimized before execution. The figure also demonstrates key system-level interactions, including memory management and device placement, showing how the static graph approach enables comprehensive pre-execution analysis and resource allocation.

Layers and Tensors. Modern machine learning frameworks implement neural network computations through two key abstractions: layers and tensors. Layers represent computational units that perform operations like convolution, pooling, or dense transformations. Each layer maintains internal states, including weights and biases, that evolve during model training. When data flows through these layers, it takes the form of tensors—immutable mathematical objects that hold and transmit numerical values.

The relationship between layers and tensors mirrors the distinction between operations and data in traditional programming. A layer defines how to transform input tensors into output tensors, much like a function defines how to

[7] Computation graphs are used to visualize the sequence of operations in a given model and to facilitate automatic differentiation which trains models through backpropagation.

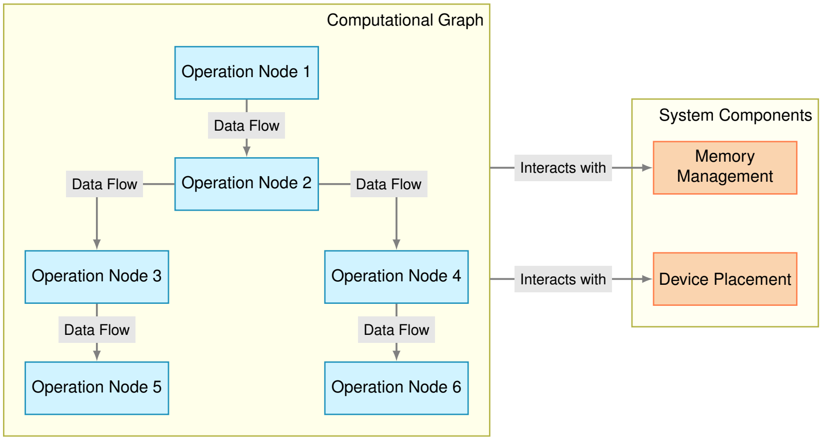


Figure 7.5: Example of a computational graph.

transform its inputs into outputs. However, layers add an extra dimension: they maintain and update internal parameters during training. For example, a convolutional layer not only specifies how to perform convolution operations but also learns and stores the optimal convolution filters for a given task.

Frameworks like TensorFlow and PyTorch leverage this abstraction to simplify model implementation. When a developer writes `tf.keras.layers. Conv2D`, the framework constructs the necessary graph nodes for convolution operations, parameter management, and data flow. This high-level interface shields developers from the complexities of implementing convolution operations, managing memory, or handling parameter updates during training.

Neural Network Construction. The power of computational graphs extends beyond basic layer operations. Activation functions, essential for introducing non-linearity in neural networks, become nodes in the graph. Functions like ReLU, sigmoid, and tanh transform the output tensors of layers, enabling networks to approximate complex mathematical functions. Frameworks provide optimized implementations of these activation functions, allowing developers to experiment with different non-linearities without worrying about implementation details.

Modern frameworks further extend this abstraction by providing complete model architectures as pre-configured computational graphs. Models like ResNet and MobileNet, which have proven effective across many tasks, come ready to use. Developers can start with these architectures, customize specific layers for their needs, and leverage transfer learning from pre-trained weights. This approach accelerates development while maintaining the benefits of carefully optimized implementations.

System-Level Consequences. The computational graph abstraction fundamentally shapes how machine learning frameworks operate. By representing computations as a directed acyclic graph, frameworks gain the ability to analyze and optimize the entire computation before execution begins. The explicit representation of data dependencies enables automatic differentiation—a crucial capability for training neural networks through gradient-based optimization.

This graph structure also provides flexibility in execution. The same model definition can run efficiently across different hardware platforms, from CPUs to GPUs to specialized accelerators. The framework handles the complexity of mapping operations to specific hardware capabilities, optimizing memory usage, and coordinating parallel execution. Moreover, the graph structure enables model serialization, allowing trained models to be saved, shared, and deployed across different environments.

While neural network diagrams help visualize model architecture, computational graphs serve a deeper purpose. They provide the precise mathematical representation needed to bridge the gap between intuitive model design and efficient execution. Understanding this representation reveals how frameworks transform high-level model descriptions into optimized, hardware-specific implementations, making modern deep learning practical at scale.

It is important to differentiate computational graphs from neural network diagrams, such as those for multilayer perceptrons (MLPs), which depict nodes and layers. Neural network diagrams visualize the architecture and flow of data through nodes and layers, providing an intuitive understanding of the model's structure. In contrast, computational graphs provide a low-level representation of the underlying mathematical operations and data dependencies required to implement and train these networks.

From a systems perspective, computational graphs provide several key capabilities that influence the entire machine learning pipeline. They enable automatic differentiation[8], which we will discuss later, provide clear structure for analyzing data dependencies and potential parallelism, and serve as an intermediate representation that can be optimized and transformed for different hardware targets. Understanding this architecture is essential for comprehending how frameworks translate high-level model descriptions into efficient executable code.

7.3.1.2 Static Graphs

Static computation graphs, pioneered by early versions of TensorFlow, implement a "define-then-run" execution model. In this approach, developers must specify the entire computation graph before execution begins. This architectural choice has significant implications for both system performance and development workflow, as we will examine later.

A static computation graph implements a clear separation between the definition of operations and their execution. During the definition phase, each mathematical operation, variable, and data flow connection is explicitly declared and added to the graph structure. This graph is a complete specification of the computation but does not perform any actual calculations. Instead, the framework constructs an internal representation of all operations and their dependencies, which will be executed in a subsequent phase.

This upfront definition enables powerful system-level optimizations. The framework can analyze the complete structure to identify opportunities for operation fusion, eliminating unnecessary intermediate results. Memory requirements can be precisely calculated and optimized in advance, leading to

[8] A computational technique that systematically computes derivatives of functions using the chain rule, crucial for training machine learning models through gradient-based optimization.

efficient allocation strategies. Furthermore, static graphs can be compiled into highly optimized executable code for specific hardware targets, taking full advantage of platform-specific features. Once validated, the same computation can be run repeatedly with high confidence in its behavior and performance characteristics.

Figure 7.6 illustrates this fundamental two-phase approach: first, the complete computational graph is constructed and optimized; then, during the execution phase, actual data flows through the graph to produce results. This separation enables the framework to perform comprehensive analysis and optimization of the entire computation before any execution begins.

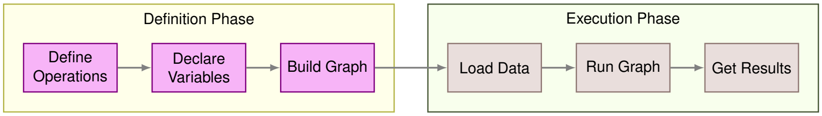


Figure 7.6: The two-phase execution model of static computation graphs.

7.3.1.3 Dynamic Graphs

Dynamic computation graphs, popularized by PyTorch, implement a "define-by-run" execution model. This approach constructs the graph during execution, offering greater flexibility in model definition and debugging. Unlike static graphs, which rely on predefined memory allocation, dynamic graphs allocate memory as operations execute, making them susceptible to memory fragmentation[9] in long-running tasks.

As shown in Figure 7.7, each operation is defined, executed, and completed before moving on to define the next operation. This contrasts sharply with static graphs, where all operations must be defined upfront. When an operation is defined, it is immediately executed, and its results become available for subsequent operations or for inspection during debugging. This cycle continues until all operations are complete.

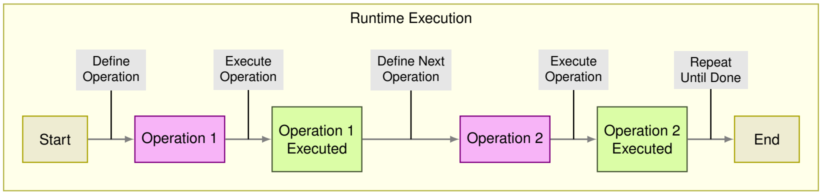


Figure 7.7: Dynamic graph execution model, illustrating runtime graph construction and immediate execution.

Dynamic graphs excel in scenarios that require conditional execution or dynamic control flow, such as when processing variable-length sequences or implementing complex branching logic. They provide immediate feedback during development, making it easier to identify and fix issues in the computational pipeline. This flexibility aligns naturally with imperative programming patterns familiar to most developers, allowing them to inspect and modify computations at runtime. These characteristics make dynamic graphs particularly valuable during the research and development phase of ML projects.

[9] Memory Fragmentation: The inefficient use of memory caused by small, unused gaps between allocated memory blocks, often resulting in wasted memory or reduced performance.

7.3.1.4 System Consequences

The architectural differences between static and dynamic computational graphs have multiple implications for how machine learning systems are designed and executed. These implications touch on various aspects of memory usage, device utilization, execution optimization, and debugging, all of which play crucial roles in determining the efficiency and scalability of a system. Here, we start with a focus on memory management and device placement as foundational concepts, leaving more detailed discussions for later chapters. This allows us to build a clear understanding before exploring more complex topics like optimization and fault tolerance.

Memory Management. Memory management occurs when executing computational graphs. Static graphs benefit from their predefined structure, allowing for precise memory planning before execution. Frameworks can calculate memory requirements in advance, optimize allocation, and minimize overhead through techniques like memory reuse. This structured approach helps ensure consistent performance, particularly in resource-constrained environments, such as Mobile and Tiny ML systems.

Dynamic graphs, by contrast, allocate memory dynamically as operations are executed. While this flexibility is invaluable for handling dynamic control flows or variable input sizes, it can result in higher memory overhead and fragmentation. These trade-offs are often most apparent during development, where dynamic graphs enable rapid iteration and debugging but may require additional optimization for production deployment.

Device Placement. Device placement, the process of assigning operations to hardware resources such as CPUs, GPUs, or specialized ASICS like TPUs, is another system-level consideration. Static graphs allow for detailed pre-execution analysis, enabling the framework to map computationally intensive operations efficiently to devices while minimizing communication overhead. This capability makes static graphs well-suited for optimizing execution on specialized hardware, where performance gains can be significant.

Dynamic graphs, in contrast, handle device placement at runtime. This allows them to adapt to changing conditions, such as hardware availability or workload demands. However, the lack of a complete graph structure before execution can make it challenging to optimize device utilization fully, potentially leading to inefficiencies in large-scale or distributed setups.

Broader Perspective. The trade-offs between static and dynamic graphs extend well beyond memory and device considerations. As shown in Table 7.1, these architectures influence optimization potential, debugging capabilities, scalability, and deployment complexity. While these broader implications are not the focus of this section, they will be explored in detail in later chapters, particularly in the context of training workflows and system-level optimizations.

These hybrid solutions aim to provide the flexibility of dynamic graphs during development while enabling the performance optimizations of static graphs in production environments. The choice between static and dynamic graphs often depends on specific project requirements, balancing factors like development speed, production performance, and system complexity.

Table 7.1: Comparison of static and dynamic computational graphs.

Aspect	Static Graphs	Dynamic Graphs
Memory Management	Precise allocation planning, optimized memory usage	Flexible but potentially less efficient allocation
Optimization Potential	Comprehensive graph-level optimizations possible	Limited to local optimizations due to runtime construction
Hardware Utilization	Can generate highly optimized hardware-specific code	May sacrifice some hardware-specific optimizations
Development Experience	Requires more upfront planning, harder to debug	Better debugging, faster iteration cycles
Runtime Flexibility	Fixed computation structure	Can adapt to runtime conditions
Production Performance	Generally better performance at scale	May have overhead from runtime graph construction
Integration with Traditional Code	More separation between definition and execution	Natural integration with imperative code
Memory Overhead	Lower memory overhead due to planned allocations	Higher memory overhead due to dynamic allocations
Debugging Capability	Limited to pre-execution analysis	Runtime inspection and modification possible
Deployment Complexity	Simpler deployment due to fixed structure	May require additional runtime support

7.3.2 Automatic Differentiation

Machine learning frameworks must solve a fundamental computational challenge: calculating derivatives through complex chains of mathematical operations efficiently and accurately. This capability enables the training of neural networks by computing how millions of parameters should be adjusted to improve the model's performance (Baydin et al. 2017b).

Listing 7.1 shows a simple computation that illustrates this challenge.

Listing 7.1: Illustrating the need for automatic differentiation

```python
def f(x):
    a = x * x       # Square
    b = sin(x)      # Sine
    return a * b    # Product
```

Even in this basic example, computing derivatives manually would require careful application of calculus rules - the product rule, the chain rule, and derivatives of trigonometric functions. Now imagine scaling this to a neural network with millions of operations. This is where automatic differentiation (AD) becomes essential.

Automatic differentiation calculates derivatives of functions implemented as computer programs by decomposing them into elementary operations. In our example, AD breaks down `f(x)` into three basic steps:

1. Computing `a = x * x` (squaring)
2. Computing `b = sin(x)` (sine function)
3. Computing the final product `a * b`

For each step, AD knows the basic derivative rules:

- For squaring: $d(x^2)/dx = 2x$

- For sine: `d(sin(x))/dx = cos(x)`
- For products: `d(uv)/dx = u(dv/dx) + v(du/dx)`

By tracking how these operations combine and systematically applying the chain rule, AD computes exact derivatives through the entire computation. When implemented in frameworks like PyTorch or TensorFlow, this enables automatic computation of gradients through arbitrary neural network architectures.[10] This fundamental understanding of how AD decomposes and tracks computations sets the foundation for examining its implementation in machine learning frameworks. We will explore its mathematical principles, system architecture implications, and performance considerations that make modern machine learning possible.

7.3.2.1 Computational Methods

Forward Mode. Forward mode automatic differentiation computes derivatives alongside the original computation, tracking how changes propagate from input to output. This approach mirrors how we might manually compute derivatives, making it intuitive to understand and implement in machine learning frameworks.

Consider our previous example with a slight modification to show how forward mode works (see Listing 7.2).

Listing 7.2: Forward mode automatic differentiation in practice

```
def f(x):    # Computing both value and derivative
  # Step 1: x -> x²
  a = x * x            # Value: x²
  da = 2 * x           # Derivative: 2x

  # Step 2: x -> sin(x)
  b = sin(x)           # Value: sin(x)
  db = cos(x)          # Derivative: cos(x)

  # Step 3: Combine using product rule
  result = a * b       # Value: x² * sin(x)
  dresult = a * db + b * da  # Derivative:
                             # x²*cos(x) + sin(x)*2x

  return result, dresult
```

Forward mode achieves this systematic derivative computation by augmenting each number with its derivative value, creating what mathematicians call a “dual number.” The example in Listing 7.3 shows how this works numerically when x = 2.0, the computation tracks both values and derivatives:

Implementation Structure. Forward mode AD structures computations to track both values and derivatives simultaneously through programs. The structure

[10] Automatic differentiation (AD) benefits diverse fields beyond machine learning, including physics simulations, design optimization, and financial risk analysis, by efficiently and accurately computing derivatives for complex processes [@paszke2019].

Listing 7.3: Forward mode with dual numbers

```
x = 2.0    # Initial value
dx = 1.0   # We're tracking derivative with respect to x

# Step 1: x²
a = 4.0    # (2.0)²
da = 4.0   # 2 * 2.0

# Step 2: sin(x)
b = 0.909   # sin(2.0)
db = -0.416 # cos(2.0)

# Final result
result = 3.637   # 4.0 * 0.909
dresult = 2.805  # 4.0 * (-0.416) + 0.909 * 4.0
```

of such computations can be seen again in Listing 7.4, where each intermediate operation is made explicit.

Listing 7.4: Structured view of a computation for forward mode AD

```
def f(x):
    a = x * x
    b = sin(x)
    return a * b
```

When a framework executes this function in forward mode, it augments each computation to carry two pieces of information: the value itself and how that value changes with respect to the input. This paired movement of value and derivative mirrors how we think about rates of change as shown in Listing 7.5.

Listing 7.5: Dual tracking of values and derivatives in forward mode AD

```
# Conceptually, each computation tracks (value, derivative)
x = (2.0, 1.0)           # Input value and its derivative
a = (4.0, 4.0)           # x² and its derivative 2x
b = (0.909, -0.416)      # sin(x) and its derivative cos(x)
result = (3.637, 2.805)  # Final value and derivative
```

This forward propagation of derivative information happens automatically within the framework's computational machinery. The framework: 1. Enriches each value with derivative information 2. Transforms each basic operation to handle both value and derivative 3. Propagates this information forward through the computation

The beauty of this approach is that it follows the natural flow of computation - as values move forward through the program, their derivatives move with them. This makes forward mode particularly well-suited for functions with single inputs and multiple outputs, as the derivative information follows the same path as the regular computation.

Performance Characteristics. Forward mode AD exhibits distinct performance patterns that influence when and how frameworks employ it. Understanding these characteristics helps explain why frameworks choose different AD approaches for different scenarios.

Forward mode performs one derivative computation alongside each original operation. For a function with one input variable, this means roughly doubling the computational work - once for the value, once for the derivative. The cost scales linearly with the number of operations in the program, making it predictable and manageable for simple computations.

However, consider a neural network layer computing derivatives for matrix multiplication between weights and inputs. To compute derivatives with respect to all weights, forward mode would need to perform the computation once for each weight parameter - potentially thousands of times. This reveals an important characteristic: forward mode's efficiency depends on the number of input variables we need derivatives for.

Forward mode's memory requirements are relatively modest. It needs to store the original value, a single derivative value, and temporary results during computation. The memory usage stays constant regardless of how complex the computation becomes. This predictable memory pattern makes forward mode particularly suitable for embedded systems with limited memory, real-time applications requiring consistent memory use, and systems where memory bandwidth is a bottleneck.

This combination of computational scaling with input variables but constant memory usage creates specific trade-offs that influence framework design decisions. Forward mode shines in scenarios with few inputs but many outputs, where its straightforward implementation and predictable resource usage outweigh the computational cost of multiple passes.

Use Cases. While forward mode automatic differentiation isn't the primary choice for training full neural networks, it plays several important roles in modern machine learning frameworks. Its strength lies in scenarios where we need to understand how small changes in inputs affect a network's behavior. Consider a data scientist trying to understand why their model makes certain predictions. They might want to analyze how changing a single pixel in an image or a specific feature in their data affects the model's output, as illustrated in Listing 7.6.

As the computation moves through each layer, forward mode carries both values and derivatives, making it straightforward to see how input perturbations ripple through to the final prediction. For each operation, we can track exactly how small changes propagate forward.

Listing 7.6: Sensitivity analysis using forward mode AD in a neural network

```python
def analyze_image_sensitivity(model, image):
    # Forward mode tracks how changing one pixel
    # affects the final classification
    layer1 = relu(W1 @ image + b1)
    layer2 = relu(W2 @ layer1 + b2)
    predictions = softmax(W3 @ layer2 + b3)
    return predictions
```

Neural network interpretation presents another compelling application. When researchers want to generate saliency maps or attribution scores, they often need to compute how each input element influences the output as shown in Listing 7.7.

Listing 7.7: Feature importance analysis using forward mode AD

```python
def compute_feature_importance(model, input_features):
    # Track influence of each input feature
    # through the network's computation
    hidden = tanh(W1 @ input_features + b1)
    logits = W2 @ hidden + b2
    # Forward mode efficiently computes d(logits)/d(input)
    return logits
```

In specialized training scenarios, particularly those involving online learning where models update on individual examples, forward mode offers advantages. The framework can track derivatives for a single example through the network efficiently, though this approach becomes less practical when dealing with batch training or updating multiple model parameters simultaneously.

Understanding these use cases helps explain why machine learning frameworks maintain forward mode capabilities alongside other differentiation strategies. While reverse mode handles the heavy lifting of full model training, forward mode provides an elegant solution for specific analytical tasks where its computational pattern matches the problem structure.

Reverse Mode. Reverse mode automatic differentiation forms the computational backbone of modern neural network training. This isn't by accident - reverse mode's structure perfectly matches what we need for training neural networks. During training, we have one scalar output (the loss function) and need derivatives with respect to millions of parameters (the network weights). Reverse mode is exceptionally efficient at computing exactly this pattern of derivatives.

A closer look at Listing 7.8 reveals how reverse mode differentiation is structured.

Listing 7.8: Basic example of reverse mode automatic differentiation

```
def f(x):
    a = x * x            # First operation: square x
    b = sin(x)           # Second operation: sine of x
    c = a * b            # Third operation: multiply results
    return c
```

In this function shown in Listing 7.8, we have three operations that create a computational chain. Notice how ‘x’ influences the final result ‘c’ through two different paths: once through squaring (a = x^2) and once through sine (b = sin(x)). We’ll need to account for both paths when computing derivatives.

First, the forward pass computes and stores values, as illustrated in Listing 7.9.

Listing 7.9: Forward pass: computing and storing each intermediate value

```
 x = 2.0             # Our input value
 a = 4.0             # x * x = 2.0 * 2.0 = 4.0
 b = 0.909           # sin(2.0)   0.909
 c = 3.637           # a * b = 4.0 * 0.909   3.637
```

Then comes the backward pass. This is where reverse mode shows its elegance. This process is demonstrated in Listing 7.10, where we compute the gradient starting from the output.

Listing 7.10: Backward pass: computing gradients through multiple paths

```
 dc/dc = 1.0    # Derivative of output with respect
                # to itself is 1

 # Moving backward through multiplication c = a * b
 dc/da = b      #  (a*b)/ a = b = 0.909
 dc/db = a      #  (a*b)/ b = a = 4.0

 # Finally, combining derivatives for x through both paths
 # Path 1: x -> x² -> c    contribution: 2x * dc/da
 # Path 2: x -> sin(x) -> c contribution: cos(x) * dc/db
 dc/dx = (2 * x * dc/da) + (cos(x) * dc/db)
       = (2 * 2.0 * 0.909) + (cos(2.0) * 4.0)
       = 3.636 + (-0.416 * 4.0)
       = 2.805
```

The power of reverse mode becomes clear when we consider what would happen if we added more operations that depend on x. Forward mode would need to track derivatives through each new path, but reverse mode efficiently handles all paths in a single backward pass. This is exactly the scenario in neural networks, where each weight can affect the final loss through multiple paths in the network.

Implementation Structure. The implementation of reverse mode in machine learning frameworks requires careful orchestration of computation and memory. While forward mode simply augments each computation, reverse mode needs to maintain a record of the forward computation to enable the backward pass. Modern frameworks accomplish this through computational graphs and automatic gradient accumulation.

Let's extend our previous example to a small neural network computation — see Listing 7.11 for the code structure.

Listing 7.11: Reverse mode applied to a simple neural network computation

```
def simple_network(x, w1, w2):
    # Forward pass
    hidden = x * w1                # First layer multiplication
    activated = max(0, hidden)     # ReLU activation
    output = activated * w2        # Second layer multiplication
    return output                  # Final output (before loss)
```

During the forward pass, the framework doesn't just compute values — it builds a graph of operations while tracking intermediate results, as illustrated in Listing 7.12.

Listing 7.12: Tracked forward pass

```
x = 1.0
w1 = 2.0
w2 = 3.0

hidden = 2.0          # x * w1 = 1.0 * 2.0
activated = 2.0       # max(0, 2.0) = 2.0
output = 6.0          # activated * w2 = 2.0 * 3.0
```

Refer to Listing 7.13 for a step-by-step breakdown of gradient computation during the backward pass.

This example illustrates several key implementation considerations: 1. The framework must track dependencies between operations 2. Intermediate values must be stored for the backward pass 3. Gradient computations follow the reverse topological order of the forward computation 4. Each operation needs both forward and backward implementations

Memory Management Strategies. Memory management represents one of the key challenges in implementing reverse mode differentiation in machine learning

Listing 7.13: Backward pass using stored values

```
d_output = 1.0          # Start with derivative of output

d_w2 = activated        # d_output * d(output)/d_w2
                        # = 1.0 * 2.0 = 2.0
d_activated = w2        # d_output * d(output)/d_activated
                        # = 1.0 * 3.0 = 3.0

# ReLU gradient: 1 if input was > 0, 0 otherwise
d_hidden = d_activated * (1 if hidden > 0 else 0)
# 3.0 * 1 = 3.0

d_w1 = x * d_hidden    # 1.0 * 3.0 = 3.0
d_x = w1 * d_hidden    # 2.0 * 3.0 = 6.0
```

frameworks. Unlike forward mode where we can discard intermediate values as we go, reverse mode requires storing results from the forward pass to compute gradients during the backward pass.

This requirement is illustrated in Listing 7.14, which extends our neural network example to highlight how intermediate activations must be preserved for use during backpropagation.

Listing 7.14: Tracking intermediate values in reverse mode

```
def deep_network(x, w1, w2, w3):
    # Forward pass - must store intermediates
    hidden1 = x * w1
    activated1 = max(0, hidden1)   # Store for backward
    hidden2 = activated1 * w2
    activated2 = max(0, hidden2)   # Store for backward
    output = activated2 * w3
    return output
```

Each intermediate value needed for gradient computation must be kept in memory until its backward pass completes. As networks grow deeper, this memory requirement grows linearly with network depth. For a typical deep neural network processing a batch of images, this can mean gigabytes of stored activations.

Frameworks employ several strategies to manage this memory burden. One such approach is illustrated in Listing 7.15.

Modern frameworks automatically balance memory usage and computation speed. They might recompute some intermediate values during the backward pass rather than storing everything, particularly for memory-intensive oper-

Listing 7.15: Conceptual example of memory management

```
def training_step(model, input_batch):
    # Strategy 1: Checkpointing
    with checkpoint_scope():
        hidden1 = activation(layer1(input_batch))
        # Framework might free some memory here
        hidden2 = activation(layer2(hidden1))
        # More selective memory management
        output = layer3(hidden2)

    # Strategy 2: Gradient accumulation
    loss = compute_loss(output)
    # Backward pass with managed memory
    loss.backward()
```

ations. This trade-off between memory and computation becomes especially important in large-scale training scenarios.

Optimization Techniques. Reverse mode automatic differentiation in machine learning frameworks employs several key optimization techniques to enhance training efficiency. These optimizations become crucial when training large neural networks where computational and memory resources are pushed to their limits.

Modern frameworks implement gradient checkpointing, a technique that strategically balances computation and memory. A simplified forward pass of such a network is shown in Listing 7.16.

Listing 7.16: Simplified forward pass in a deep neural network

```
def deep_network(input_tensor):
    # A typical deep network computation
    layer1 = large_dense_layer(input_tensor)
    activation1 = relu(layer1)
    layer2 = large_dense_layer(activation1)
    activation2 = relu(layer2)
    # ... many more layers
    output = final_layer(activation_n)
    return output
```

Instead of storing all intermediate activations, frameworks can strategically recompute certain values during the backward pass. Listing 7.17 demonstrates how frameworks achieve this memory saving. The framework might save activations only every few layers.

Another crucial optimization involves operation fusion. Rather than treating each mathematical operation separately, frameworks combine operations that

Listing 7.17: Selective activation storage via checkpointing

```
# Conceptual representation of checkpointing
checkpoint1 = save_for_backward(activation1)
# Intermediate activations can be recomputed
checkpoint2 = save_for_backward(activation4)
# Framework balances storage vs recomputation
```

commonly occur together. Matrix multiplication followed by bias addition, for instance, can be fused into a single operation, reducing memory transfers and improving hardware utilization.

The backward pass itself can be optimized by reordering computations to maximize hardware efficiency. Consider the gradient computation for a convolution layer - rather than directly translating the mathematical definition into code, frameworks implement specialized backward operations that take advantage of modern hardware capabilities.

These optimizations work together to make the training of large neural networks practical. Without them, many modern architectures would be prohibitively expensive to train, both in terms of memory usage and computation time.

7.3.2.2 Integration with Frameworks

The integration of automatic differentiation into machine learning frameworks requires careful system design to balance flexibility, performance, and usability. Modern frameworks like PyTorch and TensorFlow expose AD capabilities through high-level APIs while maintaining the sophisticated underlying machinery.

Let's examine how frameworks present AD to users. A typical example from PyTorch is shown in Listing 7.18.

While this code appears straightforward, it masks considerable complexity. The framework must:

1. Track all operations during the forward pass
2. Build and maintain the computational graph
3. Manage memory for intermediate values
4. Schedule gradient computations efficiently
5. Interface with hardware accelerators

This integration extends beyond basic training. Frameworks must handle complex scenarios like higher-order gradients, where we compute derivatives of derivatives, and mixed-precision training. The ability to compute second-order derivatives is demonstrated in Listing 7.19.

7.3.2.3 Memory Consequences

The memory demands of automatic differentiation stem from a fundamental requirement: to compute gradients during the backward pass, we must

Listing 7.18: Exposing automatic differentiation via high-level APIs in PyTorch

```
# PyTorch-style automatic differentiation
def neural_network(x):
    # Framework transparently tracks operations
    layer1 = nn.Linear(784, 256)
    layer2 = nn.Linear(256, 10)

    # Each operation is automatically tracked
    hidden = torch.relu(layer1(x))
    output = layer2(hidden)
    return output

# Training loop showing AD integration
for batch_x, batch_y in data_loader:
    optimizer.zero_grad()    # Clear previous gradients
    output = neural_network(batch_x)
    loss = loss_function(output, batch_y)

    # Framework handles all AD machinery
    loss.backward()         # Automatic backward pass
    optimizer.step()        # Parameter updates
```

Listing 7.19: Computing higher-order gradients using PyTorch's autograd

```
# Computing higher-order gradients
with torch.set_grad_enabled(True):
  # First-order gradient computation
  output = model(input)
  grad_output = torch.autograd.grad(
       output,
       model.parameters())

  # Second-order gradient computation
  grad2_output = torch.autograd.grad(
       grad_output,
       model.parameters())
```

remember what happened during the forward pass. This seemingly simple requirement creates interesting challenges for machine learning frameworks. Unlike traditional programs that can discard intermediate results as soon as they're used, AD systems must carefully preserve computational history.

This necessity is illustrated in Listing 7.20, which shows what happens during a neural network's forward pass.

Listing 7.20: Forward pass operations recorded for backward computation

```
def neural_network(x):
    # Each operation creates values we need to remember
    a = layer1(x)      # Must store for backward pass
    b = relu(a)        # Must store input to relu
    c = layer2(b)      # Must store for backward pass
    return c
```

When this network processes data, each operation creates not just its output, but also a memory obligation. The multiplication in layer1 needs to remember its inputs because computing its gradient later will require them. Even the seemingly simple relu function must track which inputs were negative to correctly propagate gradients. As networks grow deeper, these memory requirements accumulate — as seen in Listing 7.21.

This memory challenge becomes particularly interesting with deep neural networks.

Listing 7.21: Memory accumulation in deeper neural networks

```
# A deeper network shows the accumulating memory needs
hidden1 = large_matrix_multiply(input, weights1)
activated1 = relu(hidden1)
hidden2 = large_matrix_multiply(activated1, weights2)
activated2 = relu(hidden2)
output = large_matrix_multiply(activated2, weights3)
```

Each layer's computation adds to our memory burden. The framework must keep hidden1 in memory until we've computed gradients through hidden2, but after that, we can safely discard it. This creates a wave of memory usage that peaks when we start the backward pass and gradually recedes as we compute gradients.

Modern frameworks handle this memory choreography automatically. They track the lifetime of each intermediate value - how long it must remain in memory for gradient computation. When training large models, this careful memory management becomes as crucial as the numerical computations themselves. The framework frees memory as soon as it's no longer needed for gradient computation, ensuring that our memory usage, while necessarily large, remains as efficient as possible.

7.3.2.4 System Considerations

Automatic differentiation's integration into machine learning frameworks raises important system-level considerations that affect both framework design and training performance. These considerations become particularly apparent when training large neural networks where efficiency at every level matters.

As illustrated in Listing 7.22, a typical training loop handles both computation and system-level interaction.

Listing 7.22: System-level operations in a typical training loop

```python
def train_epoch(model, data_loader):
    for batch_x, batch_y in data_loader:
        # Moving data between CPU and accelerator
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        # Forward pass builds computational graph
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)

        # Backward pass computes gradients
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

This simple loop masks complex system interactions. The AD system must coordinate with multiple framework components: the memory allocator, the device manager, the operation scheduler, and the optimizer. Each gradient computation potentially triggers data movement between devices, memory allocation, and kernel launches on accelerators.

The scheduling of AD operations on modern hardware accelerators is illustrated in Listing 7.23.

Listing 7.23: Complex model with parallel computations

```python
def parallel_network(x):
    # These operations could run concurrently
    branch1 = conv_layer1(x)
    branch2 = conv_layer2(x)

    # Must synchronize for combination
    combined = branch1 + branch2
    return final_layer(combined)
```

The AD system must track dependencies not just for correct gradient computation, but also for efficient hardware utilization. It needs to determine which gradient computations can run in parallel and which must wait for others to complete. This dependency tracking extends across both forward and backward passes, creating a complex scheduling problem.

Modern frameworks handle these system-level concerns while maintaining a simple interface for users. Behind the scenes, they make sophisticated decisions about operation scheduling, memory allocation, and data movement, all while ensuring correct gradient computation through the computational graph.

7.3.2.5 Summary

Automatic differentiation systems represent an important computational abstraction in machine learning frameworks, transforming the mathematical concept of derivatives into efficient implementations. Through our examination of both forward and reverse modes, we've seen how frameworks balance mathematical precision with computational efficiency to enable training of modern neural networks.

The implementation of AD systems reveals key design patterns in machine learning frameworks. One such pattern is shown in Listing 7.24.

Listing 7.24: Simple computation revealing AD mechanisms

```
def computation(x, w):
    # Framework tracks operations
    hidden = x * w     # Stored for backward pass
    output = relu(hidden)  # Tracks activation pattern
    return output
```

This simple computation embodies several fundamental concepts:

1. Operation tracking for derivative computation
2. Memory management for intermediate values
3. System coordination for efficient execution

As shown in Listing 7.25, modern frameworks abstract these complexities behind clean interfaces while maintaining high performance.

Listing 7.25: Minimal API for automatic differentiation

```
loss = model(input)  # Forward pass tracks computation
loss.backward()      # Triggers efficient reverse mode AD
optimizer.step()     # Uses computed gradients
```

The effectiveness of automatic differentiation systems stems from their careful balance of competing demands. They must maintain sufficient computational history for accurate gradients while managing memory constraints, schedule operations efficiently while preserving correctness, and provide flexibility while optimizing performance.

Understanding these systems proves essential for both framework developers and practitioners. Framework developers must implement efficient AD to enable modern deep learning, while practitioners benefit from understanding AD's capabilities and constraints when designing and training models.

While automatic differentiation provides the computational foundation for gradient-based learning, its practical implementation depends heavily on how frameworks organize and manipulate data. This brings us to our next topic: the data structures that enable efficient computation and memory management in machine learning frameworks. These structures must not only support AD operations but also provide efficient access patterns for the diverse hardware platforms that power modern machine learning.

Looking Forward. The automatic differentiation systems we've explored provide the computational foundation for neural network training, but they don't operate in isolation. These systems need efficient ways to represent and manipulate the data flowing through them. This brings us to our next topic: the data structures that machine learning frameworks use to organize and process information.

Consider how our earlier examples handled numerical values (Listing 7.26).

Listing 7.26: Interpreting numerical values in AD computations

```python
def neural_network(x):
    hidden = w1 * x     # What exactly is x?
    activated = relu(hidden)  # How is hidden stored?
    output = w2 * activated  # What type of multiplication?
    return output
```

These operations appear straightforward, but they raise important questions. How do frameworks represent these values? How do they organize data to enable efficient computation and automatic differentiation? Most importantly, how do they structure data to take advantage of modern hardware?

The next section examines how frameworks answer these questions through specialized data structures, particularly tensors, that form the basic building blocks of machine learning computations.

7.3.3 Data Structures

Machine learning frameworks extend computational graphs with specialized data structures, bridging high-level computations with practical implementations. These data structures have two essential purposes: they provide containers for the numerical data that powers machine learning models, and they manage how this data is stored and moved across different memory spaces and devices.

While computational graphs specify the logical flow of operations, data structures determine how these operations actually access and manipulate data in memory. This dual role of organizing numerical data for model computations while handling the complexities of memory management and device placement shapes how frameworks translate mathematical operations into efficient executions across diverse computing platforms.

The effectiveness of machine learning frameworks depends heavily on their underlying data organization. While machine learning theory can be expressed through mathematical equations, turning these equations into practical implementations demands thoughtful consideration of data organization, storage, and manipulation. Modern machine learning models must process enormous amounts of data during training and inference, making efficient data access and memory usage critical across diverse hardware platforms.

A framework's data structures must excel in three key areas. First, they need to deliver high performance, supporting rapid data access and efficient memory use across different hardware. This includes optimizing memory layouts for cache efficiency and enabling smooth data transfer between memory hierarchies and devices. Second, they must offer flexibility, accommodating various model architectures and training approaches while supporting different data types and precision requirements. Third, they should provide clear and intuitive interfaces to developers while handling complex memory management and device placement behind the scenes.

These data structures bridge mathematical concepts and practical computing systems. The operations in machine learning, such as matrix multiplication, convolution, and activation functions, set basic requirements for how data must be organized. These structures must maintain numerical precision and stability while enabling efficient implementation of common operations and automatic gradient computation. However, they must also work within real-world computing constraints, dealing with limited memory bandwidth, varying hardware capabilities, and the needs of distributed computing.

The design choices made in implementing these data structures significantly influence what machine learning frameworks can achieve. Poor decisions in data structure design can result in excessive memory use, limiting model size and batch capabilities. They might create performance bottlenecks that slow down training and inference, or produce interfaces that make programming error-prone. On the other hand, thoughtful design enables automatic optimization of memory usage and computation, efficient scaling across hardware configurations, and intuitive programming interfaces that support rapid implementation of new techniques.

As we explore specific data structures in the following sections, we'll examine how frameworks address these challenges through careful design decisions and optimization approaches. This understanding proves essential for anyone working with machine learning systems, whether developing new models, optimizing existing ones, or creating new framework capabilities. We begin with tensor abstractions, the fundamental building blocks of modern machine learning frameworks, before exploring more specialized structures for parameter management, dataset handling, and execution control.

7.3.3.1 Tensors

Machine learning frameworks process and store numerical data as tensors. Every computation in a neural network, from processing input data to updating model weights, operates on tensors. Training batches of images, activation maps in convolutional networks, and parameter gradients during backpropagation all take the form of tensors. This unified representation allows frameworks to implement consistent interfaces for data manipulation and optimize operations across different hardware architectures.

Structure and Dimensionality. A tensor is a mathematical object that generalizes scalars, vectors, and matrices to higher dimensions. The dimensionality forms a natural hierarchy: a scalar is a zero-dimensional tensor containing a single value, a vector is a one-dimensional tensor containing a sequence of values, and a matrix is a two-dimensional tensor containing values arranged in rows and columns. Higher-dimensional tensors extend this pattern through nested structures; for instance, as illustrated in Figure 7.8, a three-dimensional tensor can be visualized as a stack of matrices. Therefore, vectors and matrices can be considered special cases of tensors with 1D and 2D dimensions, respectively.

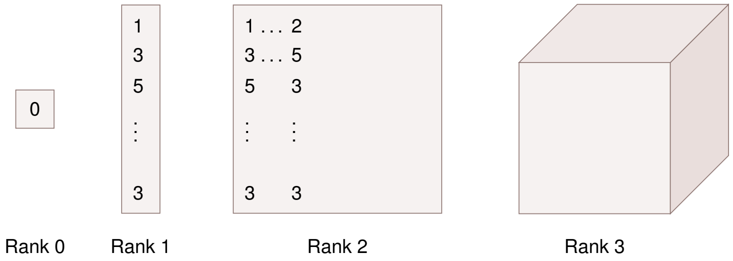


Figure 7.8: Visualization of a tensor data structure.

In practical applications, tensors naturally arise when dealing with complex data structures. As illustrated in Figure 7.9, image data exemplifies this concept particularly well. Color images comprise three channels, where each channel represents the intensity values of red, green, or blue as a distinct matrix. These channels combine to create the full colored image, forming a natural 3D tensor structure. When processing multiple images simultaneously, such as in batch operations, a fourth dimension can be added to create a 4D tensor, where each slice represents a complete three-channel image. This hierarchical organization demonstrates how tensors efficiently handle multidimensional data while maintaining clear structural relationships.

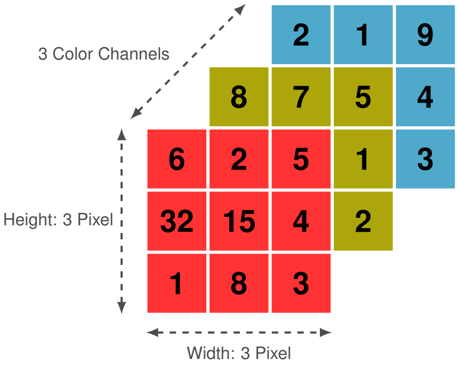


Figure 7.9: Visualization of colored image structure that can be easily stored as a 3D Tensor. Credit: Niklas Lang

In machine learning frameworks, tensors take on additional properties beyond their mathematical definition to meet the demands of modern ML systems. While mathematical tensors provide a foundation as multi-dimensional arrays with transformation properties, machine learning introduces requirements for practical computation. These requirements shape how frameworks balance mathematical precision with computational performance.

Framework tensors combine numerical data arrays with computational metadata. The dimensional structure, or shape, ranges from simple vectors and matrices to higher-dimensional arrays that represent complex data like image batches or sequence models. This dimensional information plays a critical role in operation validation and optimization. Matrix multiplication operations, for example, depend on shape metadata to verify dimensional compatibility and determine optimal computation paths.

Memory layout implementation introduces distinct challenges in tensor design. While tensors provide an abstraction of multi-dimensional data, physical computer memory remains linear. Stride patterns address this disparity by creating mappings between multi-dimensional tensor indices and linear memory addresses. These patterns significantly impact computational performance by determining memory access patterns during tensor operations. Careful alignment of stride patterns with hardware memory hierarchies maximizes cache efficiency and memory throughput.

Type Systems and Precision. Tensor implementations use type systems to control numerical precision and memory consumption. The standard choice in machine learning has been 32-bit floating-point numbers (`float32`), offering a balance of precision and efficiency. Modern frameworks extend this with multiple numeric types for different needs. Integer types support indexing and embedding operations. Reduced-precision types like 16-bit floating-point numbers enable efficient mobile deployment. 8-bit integers allow fast inference on specialized hardware.

The choice of numeric type affects both model behavior and computational efficiency. Neural network training typically requires float32 precision to maintain stable gradient computations. Inference tasks can often use lower precision (`int8` or even `int4`), reducing memory usage and increasing processing speed. Mixed-precision training[11] approaches combine these benefits by using float32 for critical accumulations while performing most computations at lower precision.

Type conversions between different numeric representations require careful management. Operating on tensors with different types demands explicit conversion rules to preserve numerical correctness. These conversions introduce computational costs and risk precision loss. Frameworks provide type casting capabilities but rely on developers to maintain numerical precision across operations.

Device Placement and Memory Management. The rise of heterogeneous computing has transformed how machine learning frameworks manage tensor operations. Modern frameworks must seamlessly operate across CPUs, GPUs, TPUs, and various other accelerators, each offering different computational advantages and memory characteristics. This diversity creates a fundamental challenge: tensors must move efficiently between devices while maintaining computational coherency throughout the execution of machine learning workloads.

[11] Mixed-precision training: A training approach that uses lower-precision arithmetic for most calculations while retaining higher-precision for critical operations, balancing performance and numerical stability.

Device placement decisions significantly influence both computational performance and memory utilization. Moving tensors between devices introduces latency costs and consumes precious bandwidth on system interconnects. Keeping multiple copies of tensors across different devices can accelerate computation by reducing data movement, but this strategy increases overall memory consumption and requires careful management of consistency between copies. Frameworks must therefore implement sophisticated memory management systems that track tensor locations and orchestrate data movement while considering these tradeoffs.

These memory management systems maintain a dynamic view of available device memory and implement strategies for efficient data transfer. When operations require tensors that reside on different devices, the framework must either move data or redistribute computation. This decision process integrates deeply with the framework's computational graph execution and operation scheduling. Memory pressure on individual devices, data transfer costs, and computational load all factor into placement decisions.

The interplay between device placement and memory management extends beyond simple data movement. Frameworks must anticipate future computational needs to prefetch data efficiently, manage memory fragmentation across devices, and handle cases where memory demands exceed device capabilities. This requires close coordination between the memory management system and the operation scheduler, especially in scenarios involving parallel computation across multiple devices or distributed training across machine boundaries.

7.3.3.2 Specialized Structures

While tensors are the building blocks of machine learning frameworks, they are not the only structures required for effective system operation. Frameworks rely on a suite of specialized data structures tailored to address the distinct needs of data processing, model parameter management, and execution coordination. These structures ensure that the entire workflow, ranging from raw data ingestion to optimized execution on hardware, proceeds seamlessly and efficiently.

Dataset Structures. Dataset structures handle the critical task of transforming raw input data into a format suitable for machine learning computations. These structures bridge the gap between diverse data sources and the tensor abstractions required by models, automating the process of reading, parsing, and preprocessing data.

Dataset structures must support efficient memory usage while dealing with input data far larger than what can fit into memory at once. For example, when training on large image datasets, these structures load images from disk, decode them into tensor-compatible formats, and apply transformations like normalization or augmentation in real time. Frameworks implement mechanisms such as data streaming, caching, and shuffling to ensure a steady supply of preprocessed batches without bottlenecks.

The design of dataset structures directly impacts training performance. Poorly designed structures can create significant overhead, limiting data throughput to GPUs or other accelerators. In contrast, well-optimized dataset handling can leverage parallelism across CPU cores, disk I/O, and memory transfers to feed accelerators at full capacity.

In large, multi-system distributed training scenarios, dataset structures also handle coordination between nodes, ensuring that each worker processes a distinct subset of data while maintaining consistency in operations like shuffling. This coordination prevents redundant computation and supports scalability across multiple devices and machines.

Parameter Structures. Parameter structures store the numerical values that define a machine learning model. These include the weights and biases of neural network layers, along with auxiliary data such as batch normalization statistics and optimizer state. Unlike datasets, which are transient, parameters persist throughout the lifecycle of model training and inference.

The design of parameter structures must balance efficient storage with rapid access during computation. For example, convolutional neural networks require parameters for filters, fully connected layers, and normalization layers, each with unique shapes and memory alignment requirements. Frameworks organize these parameters into compact representations that minimize memory consumption while enabling fast read and write operations.

A key challenge for parameter structures is managing memory efficiently across multiple devices (0003 et al. 2014). During distributed training, frameworks may replicate parameters across GPUs for parallel computation while keeping a synchronized master copy on the CPU. This strategy ensures consistency while reducing the latency of gradient updates. Additionally, parameter structures often leverage memory sharing techniques to minimize duplication, such as storing gradients and optimizer states in place to conserve memory.

Parameter structures must also adapt to various precision requirements. While training typically uses 32-bit floating-point precision for stability, reduced precision such as 16-bit floating-point or even 8-bit integers is increasingly used for inference and large-scale training. Frameworks implement type casting and mixed-precision management to enable these optimizations without compromising numerical accuracy.

Execution Structures. Execution structures coordinate how computations are performed on hardware, ensuring that operations execute efficiently while respecting device constraints. These structures work closely with computational graphs, determining how data flows through the system and how memory is allocated for intermediate results.

One of the primary roles of execution structures is memory management. During training or inference, intermediate computations such as activation maps or gradients can consume significant memory. Execution structures dynamically allocate and deallocate memory buffers to avoid fragmentation and maximize hardware utilization. For example, a deep neural network might reuse memory allocated for activation maps across layers, reducing the overall memory footprint.

These structures also handle operation scheduling, ensuring that computations are performed in the correct order and with optimal hardware utilization. On GPUs, for instance, execution structures can overlap computation and data transfer operations, hiding latency and improving throughput. When running on multiple devices, they synchronize dependent computations to maintain consistency without unnecessary delays.

Distributed training introduces additional complexity, as execution structures must manage data and computation across multiple nodes. This includes partitioning computational graphs, synchronizing gradients, and redistributing data as needed. Efficient execution structures minimize communication overhead, allowing distributed systems to scale linearly with additional hardware (B. McMahan et al. 2017a). Figure 7.10 shows how distributed training can be defined over a grid of accelerators to paralalize over multiple dimensions for faster throughput.

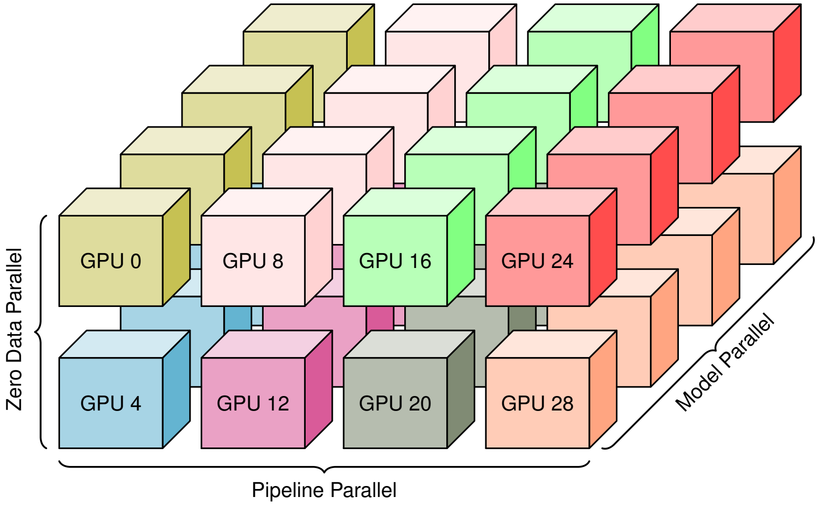


Figure 7.10: Depiction of "3D parallelism," where models can be parallelized over dimensions of devices corresponding to data replicas, sequential pipeline stages, and sharded model stages.

7.3.4 Programming Models

Programming models define how developers express computations in code. In previous sections, we explored computational graphs and specialized data structures, which together define the computational processes of machine learning frameworks. Computational graphs outline the sequence of operations, such as matrix multiplication or convolution, while data structures like tensors store the numerical values that these operations manipulate. These models fall into two categories: symbolic programming and imperative programming.

7.3.4.1 Symbolic Programming

Symbolic programming involves constructing abstract representations of computations first and executing them later. This approach aligns naturally with static computational graphs, where the entire structure is defined before any computation occurs.

For instance, in symbolic programming, variables and operations are represented as symbols. These symbolic expressions are not evaluated until explicitly executed, allowing the framework to analyze and optimize the computation graph before running it.

Consider the symbolic programming example in Listing 7.27.

Listing 7.27: Symbolic computation with delayed evaluation

```
# Expressions are constructed but not evaluated
weights = tf.Variable(tf.random.normal([784, 10]))
input = tf.placeholder(tf.float32, [None, 784])
output = tf.matmul(input, weights)

# Separate evaluation phase
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(output, feed_dict={input: data})
```

This approach enables frameworks to apply global optimizations across the entire computation, making it efficient for deployment scenarios. Additionally, static graphs can be serialized and executed across different environments, enhancing portability. Predefined graphs also facilitate efficient parallel execution strategies. However, debugging can be challenging because errors often surface during execution rather than graph construction, and modifying a static graph dynamically is cumbersome.

7.3.4.2 Imperative Programming

Imperative programming takes a more traditional approach, executing operations immediately as they are encountered. This method corresponds to dynamic computational graphs, where the structure evolves dynamically during execution.

In this programming paradigm, computations are performed directly as the code executes, closely resembling the procedural style of most general-purpose programming languages. This is demonstrated in Listing 7.28, where each operation is evaluated immediately.

Listing 7.28: Imperative programming with immediate execution

```
# Each expression evaluates immediately
weights = torch.randn(784, 10)
input = torch.randn(32, 784)
output = input @ weights  # Computation occurs now
```

The immediate execution model is intuitive and aligns with common programming practices, making it easier to use. Errors can be detected and resolved immediately during execution, simplifying debugging. Dynamic graphs allow for adjustments on-the-fly, making them ideal for tasks requiring variable graph structures, such as reinforcement learning or sequence modeling. However, the creation of dynamic graphs at runtime can introduce computational overhead, and the framework's ability to optimize the entire computation graph is limited due to the step-by-step execution process.

7.3.4.3 System Implementation Considerations

The choice between symbolic and imperative programming models fundamentally influences how ML frameworks manage system-level features such as memory management and optimization strategies.

Performance Trade-offs. In symbolic programming, frameworks can analyze the entire computation graph upfront. This allows for efficient memory allocation strategies. For example, memory can be reused for intermediate results that are no longer needed during later stages of computation. This global view also enables advanced optimization techniques such as operation fusion, automatic differentiation, and hardware-specific kernel selection. These optimizations make symbolic programming highly effective for production environments where performance is critical.

In contrast, imperative programming makes memory management and optimization more challenging since decisions must be made at runtime. Each operation executes immediately, which prevents the framework from globally analyzing the computation. This trade-off, however, provides developers with greater flexibility and immediate feedback during development. Beyond system-level features, the choice of programming model also impacts the developer experience, particularly during model development and debugging.

Development and Debugging. Symbolic programming requires developers to conceptualize their models as complete computational graphs. This often involves extra steps to inspect intermediate values, as symbolic execution defers computation until explicitly invoked. For example, in TensorFlow 1.x, developers need to use sessions and feed dictionaries to debug intermediate results, which can slow down the development process.

Imperative programming offers a more straightforward debugging experience. Operations execute immediately, allowing developers to inspect tensor values and shapes as the code runs. This immediate feedback simplifies experimentation and makes it easier to identify and fix issues in the model. As a result, imperative programming is well-suited for rapid prototyping and iterative model development.

Navigating Trade-offs. The choice between symbolic and imperative programming models often depends on the specific needs of a project. Symbolic programming excels in scenarios where performance and optimization are critical, such as production deployments. In contrast, imperative programming provides the flexibility and ease of use necessary for research and development.

Modern frameworks have introduced hybrid approaches that combine the strengths of both paradigms. For instance, TensorFlow 2.x allows developers to write code in an imperative style while converting computations into optimized graph representations for deployment. Similarly, PyTorch provides tools like TorchScript to convert dynamic models into static graphs for production use.

These hybrid approaches help bridge the gap between the flexibility of imperative programming and the efficiency of symbolic programming, enabling developers to navigate the trade-offs effectively.

7.3.5 Execution Models

Machine learning frameworks employ various execution paradigms to determine how computations are performed. These paradigms significantly influence the development experience, performance characteristics, and deployment options of ML systems. Understanding the trade-offs between execution models is essential for selecting the right approach for a given application. Let's explore three key execution paradigms: eager execution, graph execution, and just-in-time (JIT) compilation.

7.3.5.1 Eager Execution

Eager execution is the most straightforward and intuitive execution paradigm. In this model, operations are executed immediately as they are called in the code. This approach closely mirrors the way traditional imperative programming languages work, making it familiar to many developers.

Listing 7.29 demonstrates eager execution, where operations are evaluated immediately.

Listing 7.29: Eager execution in TensorFlow 2.x

```python
import tensorflow as tf

x = tf.constant([[1., 2.], [3., 4.]])
y = tf.constant([[1, 2], [3, 4]])
z = tf.matmul(x, y)
print(z)
```

In this code snippet, each line is executed sequentially. When we create the tensors `x` and `y`, they are immediately instantiated in memory. The matrix multiplication `tf.matmul(x, y)` is computed right away, and the result is stored in `z`. When we print `z`, we see the output of the computation immediately.

Eager execution offers several advantages. It provides immediate feedback, allowing developers to inspect intermediate values easily. This makes debugging more straightforward and intuitive. It also allows for more dynamic and flexible code structures, as the computation graph can change with each execution.

However, eager execution has its trade-offs. Since operations are executed immediately, the framework has less opportunity to optimize the overall computation graph. This can lead to lower performance compared to more optimized execution paradigms, especially for complex models or when dealing with large datasets.

Eager execution is particularly well-suited for research, interactive development, and rapid prototyping. It allows data scientists and researchers to quickly iterate on their ideas and see results immediately. Many modern ML frameworks, including TensorFlow 2.x and PyTorch, use eager execution as their default mode due to its developer-friendly nature.

7.3.5.2 Graph Execution

Graph execution, also known as static graph execution, takes a different approach to computing operations in ML frameworks. In this paradigm, developers first define the entire computational graph, and then execute it as a separate step.

Listing 7.30 illustrates an example in TensorFlow 1.x style, which employs graph execution.

Listing 7.30: Graph execution in TensorFlow 1.x with session-based evaluation

```python
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()

# Define the graph
x = tf.placeholder(tf.float32, shape=(2, 2))
y = tf.placeholder(tf.float32, shape=(2, 2))
z = tf.matmul(x, y)

# Execute the graph
with tf.Session() as sess:
    result = sess.run(z, feed_dict={
        x: [[1., 2.], [3., 4.]],
        y: [[1, 2], [3, 4]]
    })
    print(result)
```

In this code snippet, we first define the structure of our computation. The `placeholder` operations create nodes in the graph for input data, while `tf.matmul` creates a node representing matrix multiplication. Importantly, no actual computation occurs during this definition phase.

The execution of the graph happens when we create a session and call `sess.run()`. At this point, we provide the actual input data through the `feed_dict` parameter. The framework then has the complete graph and can perform optimizations before running the computation.

Graph execution offers several advantages. It allows the framework to see the entire computation ahead of time, enabling global optimizations that can improve performance, especially for complex models. Once defined, the graph can be easily saved and deployed across different environments, enhancing portability. It's particularly efficient for scenarios where the same computation is repeated many times with different data inputs.

However, graph execution also has its trade-offs. It requires developers to think in terms of building a graph rather than writing sequential operations, which can be less intuitive. Debugging can be more challenging because errors often don't appear until the graph is executed. Additionally, implementing dynamic computations can be more difficult with a static graph.

Graph execution is well-suited for production environments where performance and deployment consistency are crucial. It is commonly used in scenarios involving large-scale distributed training and when deploying models for predictions in high-throughput applications.

7.3.5.3 Just-In-Time Compilation

Just-In-Time compilation is a middle ground between eager execution and graph execution. This paradigm aims to combine the flexibility of eager execution with the performance benefits of graph optimization.

Listing 7.31 shows how scripted functions are compiled and reused in PyTorch.

Listing 7.31: PyTorch JIT compilation with scripted function

```python
import torch

@torch.jit.script
def compute(x, y):
    return torch.matmul(x, y)

x = torch.randn(2, 2)
y = torch.randn(2, 2)

# First call compiles the function
result = compute(x, y)
print(result)

# Subsequent calls use the optimized version
result = compute(x, y)
print(result)
```

In this code snippet, we define a function `compute` and decorate it with `@torch.jit.script`. This decorator tells PyTorch to compile the function using its JIT compiler. The first time `compute` is called, PyTorch analyzes the function, optimizes it, and generates efficient machine code. This compilation process occurs just before the function is executed, hence the term “Just-In-Time”.

Subsequent calls to `compute` use the optimized version, potentially offering significant performance improvements, especially for complex operations or when called repeatedly.

JIT compilation provides a balance between development flexibility and runtime performance. It allows developers to write code in a natural, eager-style manner while still benefiting from many of the optimizations typically associated with graph execution.

This approach offers several advantages. It maintains the immediate feedback and intuitive debugging of eager execution, as most of the code still executes eagerly. At the same time, it can deliver performance improvements for critical parts of the computation. JIT compilation can also adapt to the specific data types and shapes being used, potentially resulting in more efficient code than static graph compilation.

However, JIT compilation also has some considerations. The first execution of a compiled function may be slower due to the overhead of the compilation process. Additionally, some complex Python constructs may not be easily JIT-compiled, requiring developers to be aware of what can be optimized effectively.

JIT compilation is particularly useful in scenarios where you need both the flexibility of eager execution for development and prototyping, and the performance benefits of compilation for production or large-scale training. It's commonly used in research settings where rapid iteration is necessary but performance is still a concern.

Many modern ML frameworks incorporate JIT compilation to provide developers with a balance of ease-of-use and performance optimization, as shown in Table 7.2. This balance manifests across multiple dimensions, from the learning curve that gradually introduces optimization concepts to the runtime behavior that combines immediate feedback with performance enhancements. The table highlights how JIT compilation bridges the gap between eager execution's programming simplicity and graph execution's performance benefits, particularly in areas like memory usage and optimization scope.

Table 7.2: Comparison of execution models in machine learning frameworks.

Aspect	Eager Execution	Graph Execution	JIT Compilation
Approach	Computes each operation immediately when encountered	Builds entire computation plan first, then executes	Analyzes code at runtime, creates optimized version
Memory Usage	Holds intermediate results throughout computation	Optimizes memory by planning complete data flow	Adapts memory usage based on actual execution patterns
Optimization Scope	Limited to local operation patterns	Global optimization across entire computation chain	Combines runtime analysis with targeted optimizations
Debugging Approach	Examine values at any point during computation	Must set up specific monitoring points in graph	Initial runs show original behavior, then optimizes
Speed vs Flexibility	Prioritizes flexibility over speed	Prioritizes performance over flexibility	Balances flexibility and performance

7.3.5.4 Distributed Execution

As machine learning models continue to grow in size and complexity, training them on a single device is often no longer feasible. Large models require significant computational power and memory, while massive datasets demand efficient processing across multiple machines. To address these challenges, modern AI frameworks provide built-in support for distributed execution, allowing computations to be split across multiple GPUs, TPUs, or distributed clusters. By abstracting the complexities of parallel execution, these frameworks enable practitioners to scale machine learning workloads efficiently while maintaining ease of use.

At the essence of distributed execution are two primary strategies: data parallelism and model parallelism. Data parallelism allows multiple devices to train the same model on different subsets of data, ensuring faster convergence without increasing memory requirements. Model parallelism, on the other hand, partitions the model itself across multiple devices, allowing the training of architectures too large to fit into a single device's memory. While model parallelism comes in several variations, which will be explored in later chapters, both techniques are essential for training modern machine learning models efficiently.

Data Parallelism. Data parallelism is the most widely used approach for distributed training, enabling machine learning models to scale across multiple devices while maintaining efficiency. In this method, each computing device holds an identical copy of the model but processes a unique subset of the training data, as illustrated in Figure 8.15. Once the computations are complete, the gradients computed on each device are synchronized before updating the model parameters, ensuring consistency across all copies. This approach allows models to learn from larger datasets in parallel without increasing memory requirements per device.

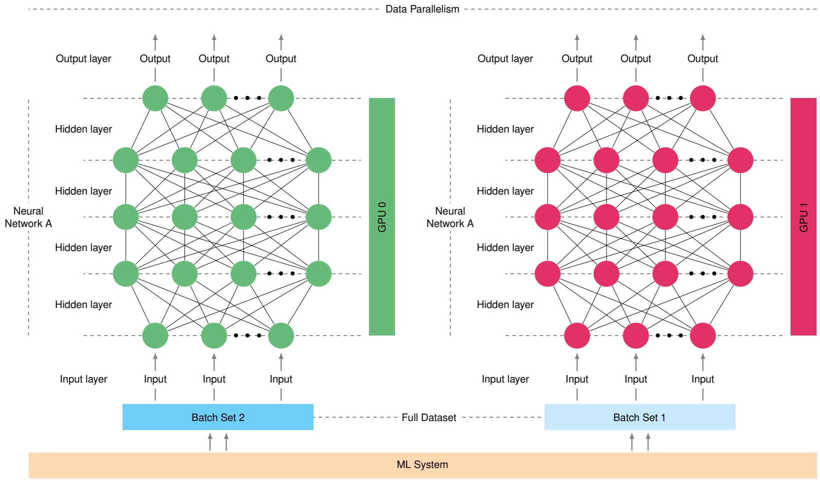


Figure 7.11: Data parallelism.

AI frameworks provide built-in mechanisms to manage the key challenges of data parallel execution, including data distribution, gradient synchronization, and performance optimization. In PyTorch, the `DistributedDataParallel (DDP)` module automates these tasks, ensuring efficient training across multiple GPUs or nodes. TensorFlow offers `tf.distribute.MirroredStrategy`, which enables seamless gradient synchronization for multi-GPU training. Similarly, JAX's `pmap()` function facilitates parallel execution across multiple accelerators, optimizing inter-device communication to reduce overhead.

By handling synchronization and communication automatically, these frameworks make distributed training accessible to a wide range of users, from researchers exploring novel architectures to engineers deploying large-scale AI systems. The implementation details vary, but the fundamental goal remains the same: enabling efficient multi-device training without requiring users to manually manage low-level parallelization.

Model Parallelism. While data parallelism is effective for many machine learning workloads, some models are too large to fit within the memory of a single device. Model parallelism addresses this limitation by partitioning the model itself across multiple devices, allowing each to process a different portion of the computation. Unlike data parallelism, where the entire model is replicated on each device, model parallelism divides layers, tensors, or specific operations among available hardware resources, as shown in Figure 8.16. This approach enables training of large-scale models that would otherwise be constrained by single-device memory limits.

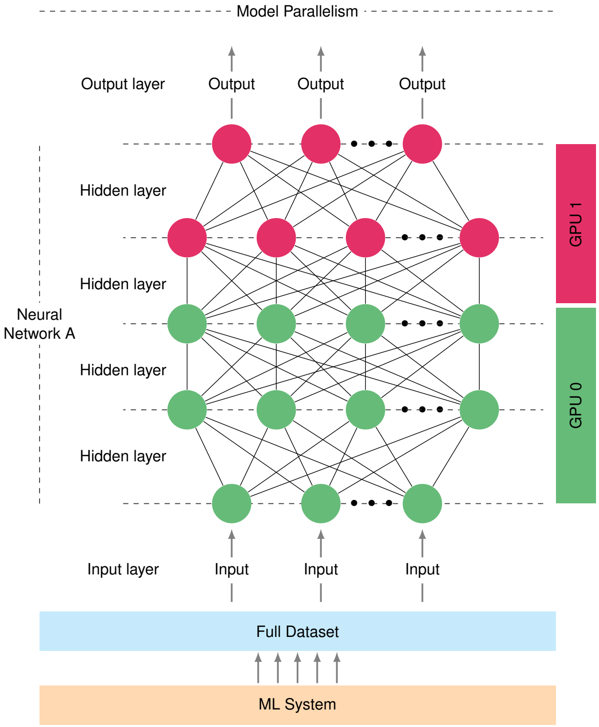


Figure 7.12: Model parallelism.

AI frameworks provide structured APIs to simplify model parallel execution, abstracting away much of the complexity associated with workload distribution and communication. PyTorch supports pipeline parallelism through `torch.distributed.pipeline.sync`, enabling different GPUs to process sequential layers of a model while maintaining efficient execution flow. TensorFlow's `TPUStrategy` allows for automatic partitioning of large models across TPU cores, optimizing execution for high-speed interconnects. Additionally, frameworks like DeepSpeed and Megatron-LM extend PyTorch by implementing advanced model sharding techniques, including tensor parallelism, which splits model weights across multiple devices to reduce memory overhead.

There are multiple variations of model parallelism, each suited to different architectures and hardware configurations. These include tensor parallelism, pipeline parallelism, and expert parallelism, among others. The specific trade-offs and applications of these techniques will be explored in later chapters, and Figure 7.13 shows some initial intuition in comparing parallelism strategies. Regardless of the exact approach, AI frameworks play an important role in managing workload partitioning, scheduling computations efficiently, and minimizing communication overhead—ensuring that even the largest models can be trained at scale.

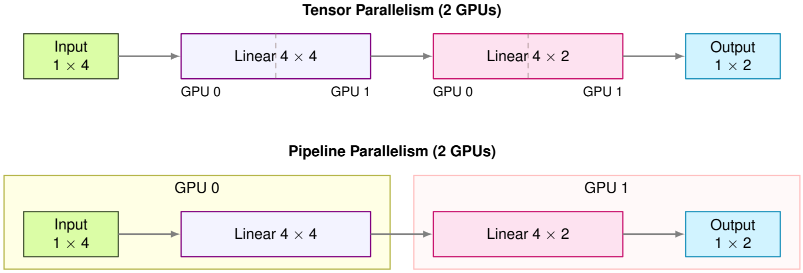


Figure 7.13: An example depiction of tensor parallelism versus pipeline parallelism. Note how the first case shards each Linear layer across GPUs, while the second assigns the first Linear to the first GPU and the second to the second GPU.

7.3.6 Core Operations

Machine learning frameworks employ multiple layers of operations that translate high-level model descriptions into efficient computations on hardware. These operations form a hierarchy: hardware abstraction operations manage the complexity of diverse computing platforms, basic numerical operations implement fundamental mathematical computations, and system-level operations coordinate resources and execution. This operational hierarchy is key to understanding how frameworks transform mathematical models into practical implementations. Figure 7.14 illustrates this hierarchy, showing the relationship between the three layers and their respective subcomponents.

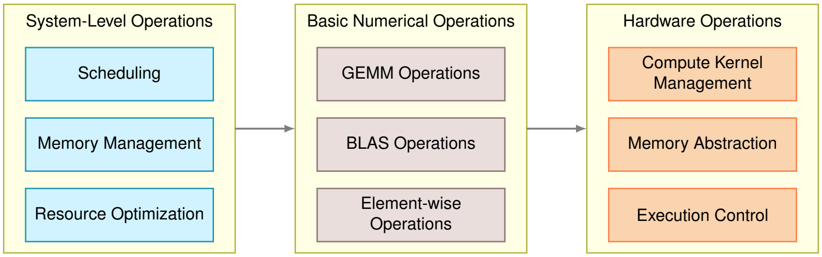


Figure 7.14: Hierarchical structure of operations in machine learning frameworks.

7.3.6.1 Hardware Abstraction Operations

At the lowest level, hardware abstraction operations provide the foundation for executing computations across diverse computing platforms. These operations isolate higher layers from hardware-specific details while maintaining computational efficiency. The abstraction layer must handle three fundamental aspects: compute kernel management, memory system abstraction, and execution control.

Compute Kernel Management. Compute kernel management involves selecting and dispatching optimal implementations of mathematical operations for different hardware architectures. This requires maintaining multiple implementations of core operations and sophisticated dispatch logic. For example, a matrix multiplication operation might be implemented using AVX-512[12] vector instructions on modern CPUs, cuBLAS on NVIDIA GPUs, or specialized tensor processing instructions on AI accelerators. The kernel manager must consider input sizes, data layout, and hardware capabilities when selecting implementations. It must also handle fallback paths for when specialized implementations are unavailable or unsuitable.

Memory System Abstraction. Memory system abstractions manage data movement through complex memory hierarchies. These abstractions must handle various memory types (registered, pinned, unified) and their specific access patterns. Data layouts often require transformation between hardware-preferred formats - for instance, between row-major and column-major matrix layouts, or between interleaved and planar image formats. The memory system must also manage alignment requirements, which can vary from 4-byte alignment on CPUs to 128-byte alignment on some accelerators. Additionally, it handles cache coherency issues when multiple execution units access the same data.

Execution Control. Execution control operations coordinate computation across multiple execution units and memory spaces. This includes managing execution queues, handling event dependencies, and controlling asynchronous operations. Modern hardware often supports multiple execution streams that can operate concurrently. For example, independent GPU streams or CPU thread pools. The execution controller must manage these streams, handle synchronization points, and ensure correct ordering of dependent operations. It must also provide error handling and recovery mechanisms for hardware-specific failures.

7.3.6.2 Basic Numerical Operations

Building upon hardware abstractions, frameworks implement fundamental numerical operations that form the building blocks of machine learning computations. These operations must balance mathematical precision with computational efficiency. General Matrix Multiply (GEMM) operations, which dominate the computational cost of most machine learning workloads. GEMM operations follow the pattern $C = \alpha AB + \beta C$, where A, B, and C are matrices, and α and β are scaling factors.

The implementation of GEMM operations requires sophisticated optimization techniques. These include blocking[13] for cache efficiency, where matrices are divided into smaller tiles that fit in cache memory; loop unrolling[14] to increase instruction-level parallelism; and specialized implementations for different matrix shapes and sparsity patterns. For example, fully-connected neural network layers typically use regular dense GEMM operations, while convolutional layers often employ specialized GEMM variants that exploit input locality patterns.

Beyond GEMM, frameworks must efficiently implement BLAS operations such as vector addition (AXPY), matrix-vector multiplication (GEMV), and

[12] A set of 512-bit single-instruction, multiple-data (SIMD) extensions to the x86 instruction set architecture.

[13] An optimization technique where computations are performed on submatrices (tiles) that fit into cache memory, reducing memory access overhead and improving computational efficiency.

[14] A method of increasing instruction-level parallelism by manually replicating loop iterations in the code, reducing branching overhead and enabling better utilization of CPU pipelines.

various reduction operations. These operations require different optimization strategies. AXPY operations are typically memory-bandwidth limited, while GEMV operations must balance memory access patterns with computational efficiency.

Element-wise operations form another critical category, including both basic arithmetic operations (addition, multiplication) and transcendental functions (exponential, logarithm, trigonometric functions). While conceptually simpler than GEMM, these operations present significant optimization opportunities through vectorization and operation fusion. For example, multiple element-wise operations can often be fused into a single kernel to reduce memory bandwidth requirements. The efficiency of these operations becomes particularly important in neural network activation functions and normalization layers, where they process large volumes of data.

Modern frameworks must also handle operations with varying numerical precision requirements. For example, training often requires 32-bit floating-point precision for numerical stability, while inference can often use reduced precision formats like 16-bit floating-point or even 8-bit integers. Frameworks must therefore provide efficient implementations across multiple numerical formats while maintaining acceptable accuracy.

7.3.6.3 System-Level Operations

System-level operations build upon the previously discussed computational graph abstractions, hardware abstractions, and numerical operations to manage overall computation flow and resource utilization. These operations handle three critical aspects: operation scheduling, memory management, and resource optimization.

Operation scheduling leverages the computational graph structure discussed earlier to determine execution ordering. Building on the static or dynamic graph representation, the scheduler must identify parallelization opportunities while respecting dependencies. The implementation challenges differ between static graphs, where the entire dependency structure is known in advance, and dynamic graphs, where dependencies emerge during execution. The scheduler must also handle advanced execution patterns like conditional operations and loops that create dynamic control flow within the graph structure.

Memory management implements sophisticated strategies for allocating and deallocating memory resources across the computational graph. Different data types require different management strategies. Model parameters typically persist throughout execution and may require specific memory types for efficient access. Intermediate results have bounded lifetimes defined by the operation graph. For example, activation values are needed only during the backward pass. The memory manager employs techniques like reference counting for automatic cleanup, memory pooling to reduce allocation overhead, and workspace management for temporary buffers. It must also handle memory fragmentation, particularly in long-running training sessions where allocation patterns can change over time.

Resource optimization integrates scheduling and memory decisions to maximize performance within system constraints. A key optimization is gradient

[15] Gradient checkpointing: A memory-saving optimization technique that stores a limited set of intermediate activations during the forward pass and recomputes the others during the backward pass to reduce memory usage.

checkpointing[15], where some intermediate results are discarded and recomputed rather than stored, trading computation time for memory savings. The optimizer must also manage concurrent execution streams, balancing load across available compute units while respecting dependencies. For operations with multiple possible implementations, it selects between alternatives based on runtime conditions - for instance, choosing between matrix multiplication algorithms based on matrix shapes and system load.

Together, these operational layers build upon the computational graph foundation to execute machine learning workloads efficiently while abstracting implementation complexity from model developers. The interaction between these layers determines overall system performance and sets the foundation for advanced optimization techniques discussed in subsequent chapters.

7.4 Framework Components

Machine learning frameworks organize their fundamental capabilities into distinct components that work together to provide a complete development and deployment environment. These components create layers of abstraction that make frameworks both usable for high-level model development and efficient for low-level execution. Understanding how these components interact helps developers choose and use frameworks effectively.

7.4.1 APIs and Abstractions

The API layer of machine learning frameworks provides the primary interface through which developers interact with the framework's capabilities. This layer must balance multiple competing demands: it must be intuitive enough for rapid development, flexible enough to support diverse use cases, and efficient enough to enable high-performance implementations.

Modern framework APIs typically implement multiple levels of abstraction. At the lowest level, they provide direct access to tensor operations and computational graph construction. These low-level APIs expose the fundamental operations discussed in the previous section, allowing fine-grained control over computation. For example, frameworks like PyTorch and TensorFlow offer such low-level interfaces, enabling researchers to define custom computations and explore novel algorithms (Paszke et al. 2019; Martín Abadi, Barham, et al. 2016), as illustrated in Listing 7.32.

Building on these primitives, frameworks implement higher-level APIs that package common patterns into reusable components. Neural network layers represent a classic example, although a convolution operation could be implemented manually using basic tensor operations, frameworks provide pre-built layer abstractions that handle the implementation details. This approach is exemplified by libraries such as PyTorch's `torch.nn` and TensorFlow's Keras API, which enable efficient and user-friendly model development (Chollet 2018), as shown in Listing 7.33.

At the highest level (Listing 7.34), frameworks often provide model-level abstractions that automate common workflows. For example, the Keras API provides a highly abstract interface that hides most implementation details:

Listing 7.32: Manual computation with PyTorch low-level API

```python
import torch

# Manual tensor operations
x = torch.randn(2, 3)
w = torch.randn(3, 4)
b = torch.randn(4)
y = torch.matmul(x, w) + b

# Manual gradient computation
y.backward(torch.ones_like(y))
```

Listing 7.33: Mid-level abstraction using PyTorch modules

```python
import torch.nn as nn

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(3, 64, kernel_size=3)
        self.fc = nn.Linear(64, 10)

    def forward(self, x):
        x = self.conv(x)
        x = torch.relu(x)
        x = self.fc(x)
        return x
```

The organization of these API layers reflects fundamental trade-offs in framework design. Lower-level APIs provide maximum flexibility but require more expertise to use effectively. Higher-level APIs improve developer productivity but may constrain implementation choices. Framework APIs must therefore provide clear paths between abstraction levels, allowing developers to mix different levels of abstraction as needed for their specific use cases.## Framework Components

Machine learning frameworks organize their fundamental capabilities into distinct components that work together to provide a complete development and deployment environment. These components create layers of abstraction that make frameworks both usable for high-level model development and efficient for low-level execution. Understanding how these components interact helps developers choose and use frameworks effectively.

Listing 7.34: High-level model definition and training with Keras

```python
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Conv2D(
        64,
        3,
        activation='relu',
        input_shape=(32, 32, 3)),
    keras.layers.Flatten(),
    keras.layers.Dense(10)
])

# Automated training workflow
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy')
model.fit(train_data, train_labels, epochs=10)
```

7.4.2 Core Libraries

At the heart of every machine learning framework lies a set of core libraries, forming the foundation upon which all other components are built. These libraries provide the essential building blocks for machine learning operations, implementing fundamental tensor operations that serve as the backbone of numerical computations. Heavily optimized for performance, these operations often leverage low-level programming languages and hardware-specific optimizations to ensure efficient execution of tasks like matrix multiplication, a cornerstone of neural network computations.

Alongside these basic operations, core libraries implement automatic differentiation capabilities, enabling the efficient computation of gradients for complex functions. This feature is crucial for the backpropagation algorithm that powers most neural network training. The implementation often involves intricate graph manipulation and symbolic computation techniques, abstracting away the complexities of gradient calculation from the end-user.

Building upon these fundamental operations, core libraries typically provide pre-implemented neural network layers such as convolutional, recurrent, and attention mechanisms. These ready-to-use components save developers from reinventing the wheel for common model architectures, allowing them to focus on higher-level model design rather than low-level implementation details. Similarly, optimization algorithms like various flavors of gradient descent are provided out-of-the-box, further streamlining the model development process.

A simplified example of how these components might be used in practice is shown in Listing 7.35.

Listing 7.35: Basic example of model training with gradient update

```
import torch
import torch.nn as nn

# Create a simple neural network
model = nn.Sequential(
    nn.Linear(10, 20),
    nn.ReLU(),
    nn.Linear(20, 1)
)

# Define loss function and optimizer
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Forward pass, compute loss, and backward pass
x = torch.randn(32, 10)
y = torch.randn(32, 1)
y_pred = model(x)
loss = loss_fn(y_pred, y)
loss.backward()
optimizer.step()
```

This example demonstrates how core libraries provide high-level abstractions for model creation, loss computation, and optimization, while handling low-level details internally.

7.4.3 Extensions and Plugins

While core libraries offer essential functionality, the true power of modern machine learning frameworks often lies in their extensibility. Extensions and plugins expand the capabilities of frameworks, allowing them to address specialized needs and leverage cutting-edge research. Domain-specific libraries, for instance, cater to particular areas like computer vision or natural language processing, providing pre-trained models, specialized data augmentation techniques, and task-specific layers.

Hardware acceleration plugins play an important role in performance optimization as it enables frameworks to take advantage of specialized hardware like GPUs or TPUs. These plugins dramatically speed up computations and allow seamless switching between different hardware backends, a key feature for scalability and flexibility in modern machine learning workflows.

As models and datasets grow in size and complexity, distributed computing extensions also become important. These tools enable training across multiple devices or machines, handling complex tasks like data parallelism, model parallelism, and synchronization between compute nodes. This capability is essential for researchers and companies tackling large-scale machine learning problems.

Complementing these computational tools are visualization and experiment tracking extensions. Visualization tools provide invaluable insights into the training process and model behavior, displaying real-time metrics and even offering interactive debugging capabilities. Experiment tracking extensions help manage the complexity of machine learning research, allowing systematic logging and comparison of different model configurations and hyperparameters.

7.4.4 Development Tools

The ecosystem of development tools surrounding a machine learning framework further enhances its effectiveness and adoption. Interactive development environments, such as Jupyter notebooks, have become nearly ubiquitous in machine learning workflows, allowing for rapid prototyping and seamless integration of code, documentation, and outputs. Many frameworks provide custom extensions for these environments to enhance the development experience.

Debugging and profiling tools address the unique challenges presented by machine learning models. Specialized debuggers allow developers to inspect the internal state of models during training and inference, while profiling tools identify bottlenecks in model execution, guiding optimization efforts. These tools are essential for developing efficient and reliable machine learning systems.

As projects grow in complexity, version control integration becomes increasingly important. Tools that allow versioning of not just code, but also model weights, hyperparameters, and training data, help manage the iterative nature of model development. This comprehensive versioning approach ensures reproducibility and facilitates collaboration in large-scale machine learning projects.

Finally, deployment utilities bridge the gap between development and production environments. These tools handle tasks like model compression, conversion to deployment-friendly formats, and integration with serving infrastructure, streamlining the process of moving models from experimental settings to real-world applications.

7.5 System Integration

System integration is about implementing machine learning frameworks in real-world environments. This section explores how ML frameworks integrate with broader software and hardware ecosystems, addressing the challenges and considerations at each level of the integration process.

7.5.1 Hardware Integration

Effective hardware integration is crucial for optimizing the performance of machine learning models. Modern ML frameworks must adapt to a diverse range of computing environments, from high-performance GPU clusters to resource-constrained edge devices.

For GPU acceleration, frameworks like TensorFlow and PyTorch provide robust support, allowing seamless utilization of NVIDIA's CUDA platform. This integration enables significant speedups in both training and inference tasks. Similarly, support for Google's TPUs in TensorFlow allows for even further acceleration of specific workloads.

In distributed computing scenarios, frameworks must efficiently manage multi-device and multi-node setups. This involves strategies for data parallelism, where the same model is replicated across devices, and model parallelism, where different parts of the model are distributed across hardware units. Frameworks like Horovod have emerged to simplify distributed training across different backend frameworks.

For edge deployment, frameworks are increasingly offering lightweight versions optimized for mobile and IoT devices. TensorFlow Lite and PyTorch Mobile, for instance, provide tools for model compression and optimization, ensuring efficient execution on devices with limited computational resources and power constraints.

7.5.2 Software Stack

Integrating ML frameworks into existing software stacks presents unique challenges and opportunities. A key consideration is how the ML system interfaces with data processing pipelines. Frameworks often provide connectors to popular big data tools like Apache Spark or Apache Beam, allowing seamless data flow between data processing systems and ML training environments.

Containerization technologies like Docker have become essential in ML workflows, ensuring consistency between development and production environments. Kubernetes has emerged as a popular choice for orchestrating containerized ML workloads, providing scalability and manageability for complex deployments.

ML frameworks must also interface with other enterprise systems such as databases, message queues, and web services. For instance, TensorFlow Serving provides a flexible, high-performance serving system for machine learning models, which can be easily integrated into existing microservices architectures.

7.5.3 Deployment Considerations

Deploying ML models to production environments involves several critical considerations. Model serving strategies must balance performance, scalability, and resource efficiency. Approaches range from batch prediction for large-scale offline processing to real-time serving for interactive applications.

Scaling ML systems to meet production demands often involves techniques like horizontal scaling of inference servers, caching of frequent predictions, and load balancing across multiple model versions. Frameworks like TensorFlow Serving and TorchServe provide built-in solutions for many of these scaling challenges.

Monitoring and logging are crucial for maintaining ML systems in production. This includes tracking model performance metrics, detecting concept drift, and logging prediction inputs and outputs for auditing purposes. Tools like Prometheus and Grafana are often integrated with ML serving systems to provide comprehensive monitoring solutions.

7.5.4 Workflow Orchestration

Managing end-to-end ML pipelines requires orchestrating multiple stages, from data preparation and model training to deployment and monitoring. MLOps practices have emerged to address these challenges, bringing DevOps principles to machine learning workflows.

Continuous Integration and Continuous Deployment (CI/CD) practices are being adapted for ML workflows. This involves automating model testing, validation, and deployment processes. Tools like Jenkins or GitLab CI can be extended with ML-specific stages to create robust CI/CD pipelines for machine learning projects.

Automated model retraining and updating is another critical aspect of ML workflow orchestration. This involves setting up systems to automatically retrain models on new data, evaluate their performance, and seamlessly update production models when certain criteria are met. Frameworks like Kubeflow provide end-to-end ML pipelines that can automate many of these processes. Figure 7.15 shows an example orchestration flow, where a user submitts DAGs, or directed acyclic graphs of workloads to process and train to be executed.

Version control for ML assets, including data, model architectures, and hyperparameters, is essential for reproducibility and collaboration. Tools like DVC (Data Version Control) and MLflow have emerged to address these ML-specific version control needs.

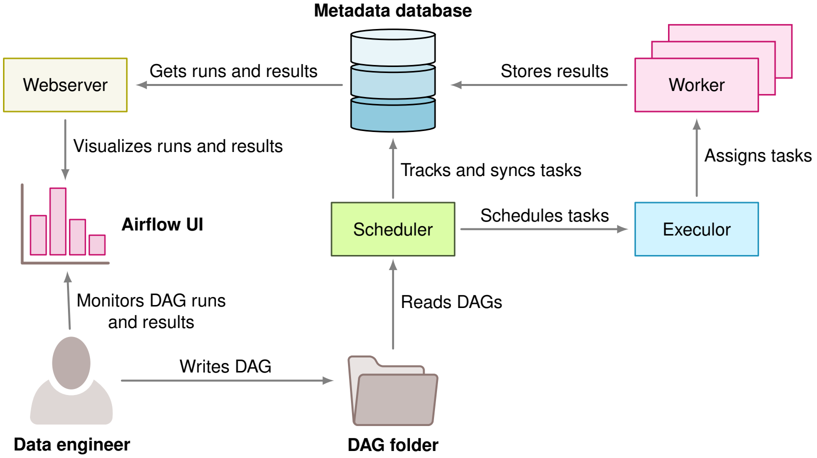


Figure 7.15: Diagram showing how a data engineer might interact with AirFlow, an example orchestration service, in scheduling tasks, executing them across distributed workers, and visualizing the results.

7.6 Major Frameworks

As we have seen earlier, machine learning frameworks are complicated. Over the years, several machine learning frameworks have emerged, each with its unique strengths and ecosystem, but few have remained as industry standards. Here we examine the mature and major players in the field, starting with a comprehensive look at TensorFlow, followed by PyTorch, JAX, and other notable frameworks.

7.6.1 TensorFlow Ecosystem

TensorFlow was developed by the Google Brain team and was released as an open-source software library on November 9, 2015. It was designed for numerical computation using data flow graphs[16] and has since become popular for a wide range of machine learning applications.

TensorFlow is a training and inference framework that provides built-in functionality to handle everything from model creation and training to deployment, as shown in Figure 7.16. Since its initial development, the TensorFlow ecosystem has grown to include many different "varieties" of TensorFlow, each intended to allow users to support ML on different platforms.

1. TensorFlow Core: primary package that most developers engage with. It provides a comprehensive, flexible platform for defining, training, and deploying machine learning models. It includes tf.keras as its high-level API.
2. TensorFlow Lite: designed for deploying lightweight models on mobile, embedded, and edge devices. It offers tools to convert TensorFlow models to a more compact format suitable for limited-resource devices and provides optimized pre-trained models for mobile.
3. TensorFlow Lite Micro: designed for running machine learning models on microcontrollers with minimal resources. It operates without the need for operating system support, standard C or C++ libraries, or dynamic memory allocation, using only a few kilobytes of memory.
4. TensorFlow.js: JavaScript library that allows training and deployment of machine learning models directly in the browser or on Node.js. It also provides tools for porting pre-trained TensorFlow models to the browser-friendly format.
5. TensorFlow on Edge Devices (Coral): platform of hardware components and software tools from Google that allows the execution of TensorFlow models on edge devices, leveraging Edge TPUs for acceleration.
6. TensorFlow Federated (TFF): framework for machine learning and other computations on decentralized data. TFF facilitates federated learning,[17] allowing model training across many devices without centralizing the data.
7. TensorFlow Graphics: library for using TensorFlow to carry out graphics-related tasks, including 3D shapes and point clouds processing, using deep learning.
8. TensorFlow Hub: repository of reusable machine learning model components to allow developers to reuse pre-trained model components, facilitating transfer learning and model composition.
9. TensorFlow Serving: framework designed for serving and deploying machine learning models for inference in production environments. It

[16] A data flow graph is a directed graph where nodes represent operations and edges represent data flowing between operations.

[17] In federated learning, multiple entities (referred to as clients) train a model on their local datasets which ensures their data remains decentralized. This technique in ML is motivated by issues such as data privacy and data minimization. The assumption that the data is independently and identically distributed is no longer valid in federated learning which may cause biased local models.

provides tools for versioning and dynamically updating deployed models without service interruption.

10. TensorFlow Extended (TFX): end-to-end platform designed to deploy and manage machine learning pipelines in production settings. TFX encompasses data validation, preprocessing, model training, validation, and serving components.

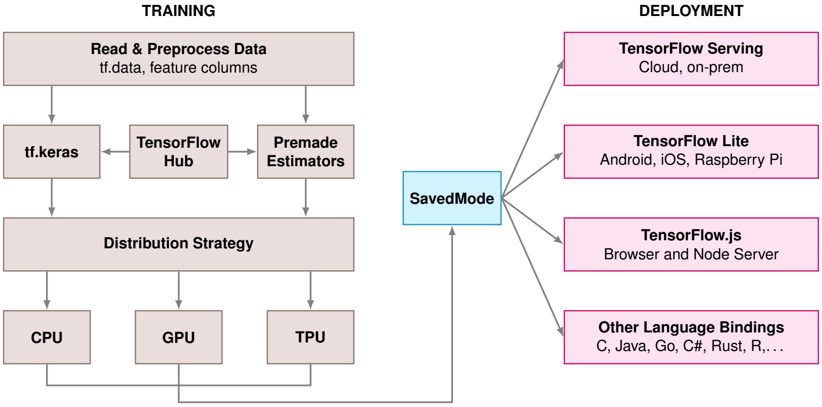


Figure 7.16: Architecture overview of TensorFlow 2.0. Source: Tensorflow.

7.6.2 PyTorch

PyTorch, developed by Facebook's AI Research lab, has gained significant traction in the machine learning community, particularly among researchers and academics. Its design philosophy emphasizes ease of use, flexibility, and dynamic computation, which aligns well with the iterative nature of research and experimentation.

PyTorch's architecture lies its dynamic computational graph system. Unlike the static graphs used in earlier versions of TensorFlow, PyTorch builds the computational graph on-the-fly during execution. This approach, often referred to as "define-by-run," allows for more intuitive model design and easier debugging that we discussed earlier. Moreover, developers can use standard Python control flow statements within their models, and the graph structure can change from iteration to iteration. This flexibility is particularly advantageous when working with variable-length inputs or complex, dynamic neural network architectures.

PyTorch's eager execution mode is tightly coupled with its dynamic graph approach. Operations are executed immediately as they are called, rather than being deferred for later execution in a static graph. This immediate execution facilitates easier debugging and allows for more natural integration with Python's native debugging tools. The eager execution model aligns closely with PyTorch's imperative programming style, which many developers find more intuitive and Pythonic.

PyTorch's fundamental data structure is the tensor, similar to TensorFlow and other frameworks discussed in earlier sections. PyTorch tensors are conceptually equivalent to multi-dimensional arrays and can be manipulated using a rich set of operations. The framework provides seamless integration with CUDA, much like TensorFlow, enabling efficient GPU acceleration for tensor computations. PyTorch's autograd system automatically tracks all operations performed on tensors, facilitating automatic differentiation for gradient-based optimization algorithms.

7.6.3 JAX

JAX, developed by Google Research, is a newer entrant in the field of machine learning frameworks. Unlike TensorFlow and PyTorch, which were primarily designed for deep learning, JAX focuses on high-performance numerical computing and advanced machine learning research. Its design philosophy centers around functional programming principles and composition of transformations, offering a fresh perspective on building and optimizing machine learning systems.

JAX is built as a NumPy-like library with added capabilities for automatic differentiation and just-in-time compilation. This foundation makes JAX feel familiar to researchers accustomed to scientific computing in Python, while providing powerful tools for optimization and acceleration. Where TensorFlow uses static computational graphs and PyTorch employs dynamic ones, JAX takes a different approach altogether, as it is a system for transforming numerical functions.

One of JAX's key features is its powerful automatic differentiation system. Unlike TensorFlow's static graph approach or PyTorch's dynamic computation, JAX can differentiate native Python and NumPy functions, including those with loops, branches, and recursion. This capability extends beyond simple scalar-to-scalar functions, allowing for complex transformations like vectorization and JIT compilation. This flexibility is particularly valuable for researchers exploring novel machine learning techniques and architectures.

JAX leverages XLA (Accelerated Linear Algebra) for just-in-time compilation, similar to TensorFlow but with a more central role in its operation. This allows JAX to optimize and compile Python code for various hardware accelerators, including GPUs and TPUs. In contrast to PyTorch's eager execution and TensorFlow's graph optimization, JAX's approach can lead to significant performance improvements, especially for complex computational patterns.

Where TensorFlow and PyTorch primarily use object-oriented and imperative programming models, JAX embraces functional programming. This approach encourages the use of pure functions and immutable data, which can lead to more predictable and easier-to-optimize code. It's a significant departure from the stateful models common in other frameworks and can require a shift in thinking for developers accustomed to TensorFlow or PyTorch.

JAX introduces a set of composable function transformations that set it apart from both TensorFlow and PyTorch. These include automatic differentiation (grad), just-in-time compilation, automatic vectorization (vmap), and parallel execution across multiple devices (pmap). These transformations can be composed, allowing for powerful and flexible operations that are not as straightforward in other frameworks.

7.6.4 Framework Comparison

Table 7.3 provides a concise comparison of three major machine learning frameworks: TensorFlow, PyTorch, and JAX. These frameworks, while serving similar purposes, exhibit fundamental differences in their design philosophies and technical implementations.

Table 7.3: Core characteristics of major machine learning frameworks.

Aspect	TensorFlow	PyTorch	JAX
Graph Type	Static (1.x), Dynamic (2.x)	Dynamic	Functional transformations
Programming Model	Imperative (2.x), Symbolic (1.x)	Imperative	Functional
Core Data Structure	Tensor (mutable)	Tensor (mutable)	Array (immutable)
Execution Mode	Eager (2.x default), Graph	Eager	Just-in-time compilation
Automatic Differentiation	Reverse mode	Reverse mode	Forward and Reverse mode
Hardware Acceleration	CPU, GPU, TPU	CPU, GPU	CPU, GPU, TPU

7.7 Framework Specialization

Machine learning frameworks have evolved significantly to meet the diverse needs of different computational environments. As ML applications expand beyond traditional data centers to encompass edge devices, mobile platforms, and even tiny microcontrollers, the need for specialized frameworks has become increasingly apparent.

Framework specialization refers to the process of tailoring ML frameworks to optimize performance, efficiency, and functionality for specific deployment environments. This specialization is crucial because the computational resources, power constraints, and use cases vary dramatically across different platforms.

Machine learning frameworks have addressed interoperability challenges through standardized model formats, with the Open Neural Network Exchange (ONNX) emerging as a widely adopted solution. ONNX defines a common representation for neural network models that enables seamless translation between different frameworks and deployment environments.

The ONNX format serves two primary purposes. First, it provides a framework-neutral specification for describing model architecture and parameters. Second, it includes runtime implementations that can execute these models across diverse hardware platforms. This standardization eliminates the need to manually convert or reimplement models when moving between frameworks.

In practice, ONNX facilitates important workflow patterns in production machine learning systems. For example, a research team might develop and train a model using PyTorch's dynamic computation graphs, then export it to ONNX for deployment using TensorFlow's production-optimized serving infrastructure. Similarly, models can be converted to ONNX format for execution on edge devices using specialized runtimes like ONNX Runtime. This interoperability, illustrated in Figure 7.17, has become increasingly important as the machine learning ecosystem has expanded. Organizations frequently need to leverage different frameworks' strengths at various stages of the machine learning lifecycle, from research and development.

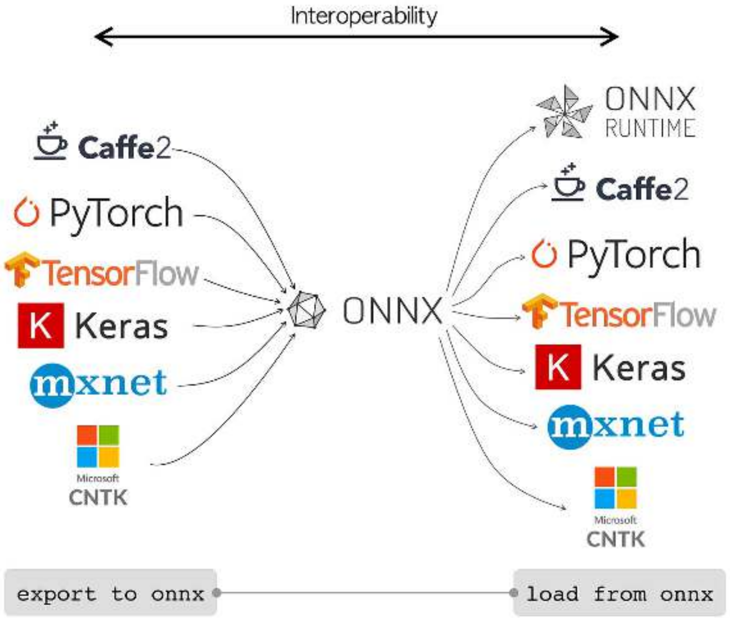


Figure 7.17: Interoperability enabled by ONNX across major ML frameworks.

Machine learning deployment environments shape how frameworks specialize and evolve. Cloud ML environments leverage high-performance servers that offer abundant computational resources for complex operations. Edge ML operates on devices with moderate computing power, where real-time processing often takes priority. Mobile ML adapts to the varying capabilities and energy constraints of smartphones and tablets. Tiny ML functions within the strict limitations of microcontrollers and other highly constrained devices that possess minimal resources.

Each of these environments presents unique challenges that influence framework design. Cloud frameworks prioritize scalability and distributed computing. Edge frameworks focus on low-latency inference and adaptability to diverse hardware. Mobile frameworks emphasize energy efficiency and integration with device-specific features. TinyML frameworks specialize in extreme resource optimization for severely constrained environments.

In the following sections, we will explore how ML frameworks adapt to each of these environments. We will examine the specific techniques and design choices that enable frameworks to address the unique challenges of each domain, highlighting the trade-offs and optimizations that characterize framework specialization.

7.7.1 Cloud-Based Frameworks

Cloud ML frameworks are sophisticated software infrastructures designed to leverage the vast computational resources available in cloud environments.

These frameworks specialize in three primary areas: distributed computing architectures, management of large-scale data and models, and integration with cloud-native services.

Distributed computing is a fundamental specialization of cloud ML frameworks. These frameworks implement advanced strategies for partitioning and coordinating computational tasks across multiple machines or graphics processing units (GPUs). This capability is essential for training large-scale models on massive datasets. Both TensorFlow and PyTorch, two leading cloud ML frameworks, offer robust support for distributed computing. TensorFlow's graph-based approach (in its 1.x version) was particularly well-suited for distributed execution, while PyTorch's dynamic computational graph allows for more flexible distributed training strategies.

The ability to handle large-scale data and models is another key specialization. Cloud ML frameworks are optimized to work with datasets and models that far exceed the capacity of single machines. This specialization is reflected in the data structures of these frameworks. For instance, both TensorFlow and PyTorch use mutable Tensor objects as their primary data structure, allowing for efficient in-place operations on large datasets. JAX, a more recent framework, uses immutable arrays, which can provide benefits in terms of functional programming paradigms and optimization opportunities in distributed settings.

Integration with cloud-native services is the third major specialization area. This integration enables automated resource scaling, seamless access to cloud storage, and incorporation of cloud-based monitoring and logging systems. The execution modes of different frameworks play a role here. TensorFlow 2.x and PyTorch both default to eager execution, which allows for easier integration with cloud services and debugging. JAX's just-in-time compilation offers potential performance benefits in cloud environments by optimizing computations for specific hardware.

Hardware acceleration is an important aspect of cloud ML frameworks. All major frameworks support CPU and GPU execution, with TensorFlow and JAX also offering native support for Google's TPU. NVIDIA's TensorRT is an optimization tool dedicated for GPU-based inference, providing sophisticated optimizations like layer fusion, precision calibration[18], and kernel auto-tuning to maximize throughput on NVIDIA GPUs. These hardware acceleration options allow cloud ML frameworks to efficiently utilize the diverse computational resources available in cloud environments.

The automatic differentiation capabilities of these frameworks are particularly important in cloud settings where complex models with millions of parameters are common. While TensorFlow and PyTorch primarily use reverse-mode differentiation, JAX's support for both forward and reverse-mode differentiation can offer advantages in certain large-scale optimization scenarios.

These specializations enable cloud ML frameworks to fully utilize the scalability and computational power of cloud infrastructure. However, this capability comes with increased complexity in deployment and management, often requiring specialized knowledge to fully leverage these frameworks. The focus on scalability and integration makes cloud ML frameworks particularly suitable

[18] A process of adjusting computations to use reduced numerical precision, balancing performance improvements with acceptable losses in accuracy.

for large-scale research projects, enterprise-level ML applications, and scenarios requiring massive computational resources.

7.7.2 Edge-Based Frameworks

Edge ML frameworks are specialized software tools designed to facilitate machine learning operations in edge computing environments, characterized by proximity to data sources, stringent latency requirements, and limited computational resources. Examples of popular edge ML frameworks include TensorFlow Lite and Edge Impulse. The specialization of these frameworks addresses three primary challenges: real-time inference optimization, adaptation to heterogeneous hardware, and resource-constrained operation.

Real-time inference optimization is a critical feature of edge ML frameworks. This often involves leveraging different execution modes and graph types. For instance, while TensorFlow Lite (the edge-focused version of TensorFlow) uses a static graph approach to optimize inference, frameworks like PyTorch Mobile maintain a dynamic graph capability, allowing for more flexible model structures at the cost of some performance. The choice between static and dynamic graphs in edge frameworks often is a trade-off between optimization potential and model flexibility.

Adaptation to heterogeneous hardware is crucial for edge deployments. Edge ML frameworks extend the hardware acceleration capabilities of their cloud counterparts but with a focus on edge-specific hardware. For instance, TensorFlow Lite supports acceleration on mobile GPUs and edge TPUs, while frameworks like ARM's Compute Library optimize for ARM-based processors. This specialization often involves custom operator implementations and low-level optimizations specific to edge hardware.

Operating within resource constraints is another aspect of edge ML framework specialization. This is reflected in the data structures and execution models of these frameworks. For instance, many edge frameworks use quantized tensors as their primary data structure, representing values with reduced precision (e.g., 8-bit integers instead of 32-bit floats) to decrease memory usage and computational demands. The automatic differentiation capabilities, while crucial for training in cloud environments, are often stripped down or removed entirely in edge frameworks to reduce model size and improve inference speed.

Edge ML frameworks also often include features for model versioning and updates, allowing for the deployment of new models with minimal system downtime. Some frameworks support limited on-device learning, enabling models to adapt to local data without compromising data privacy.

The specializations of edge ML frameworks collectively enable high-performance inference in resource-constrained environments. This capability expands the potential applications of AI in areas with limited cloud connectivity or where real-time processing is crucial. However, effective utilization of these frameworks requires careful consideration of target hardware specifications and application-specific requirements, necessitating a balance between model accuracy and resource utilization.

7.7.3 Mobile-Based Frameworks

Mobile ML frameworks are specialized software tools designed for deploying and executing machine learning models on smartphones and tablets. Examples include TensorFlow Lite and Apple's Core ML. These frameworks address the unique challenges of mobile environments, including limited computational resources, constrained power consumption, and diverse hardware configurations. The specialization of mobile ML frameworks primarily focuses on on-device inference optimization, energy efficiency, and integration with mobile-specific hardware and sensors.

On-device inference optimization in mobile ML frameworks often involves a careful balance between graph types and execution modes. For instance, TensorFlow Lite, also a popular mobile ML framework, uses a static graph approach to optimize inference performance. This contrasts with the dynamic graph capability of PyTorch Mobile, which offers more flexibility at the cost of some performance. The choice between static and dynamic graphs in mobile frameworks is a trade-off between optimization potential and model adaptability, crucial in the diverse and changing mobile environment.

The data structures in mobile ML frameworks are optimized for efficient memory usage and computation. While cloud-based frameworks like TensorFlow and PyTorch use mutable tensors, mobile frameworks often employ more specialized data structures. For example, many mobile frameworks use quantized tensors, representing values with reduced precision (e.g., 8-bit integers instead of 32-bit floats) to decrease memory footprint and computational demands. This specialization is critical given the limited RAM and processing power of mobile devices.

Energy efficiency, a paramount concern in mobile environments, influences the design of execution modes in mobile ML frameworks. Unlike cloud frameworks that may use eager execution for ease of development, mobile frameworks often prioritize graph-based execution for its potential energy savings. For instance, Apple's Core ML uses a compiled model approach, converting ML models into a form that can be efficiently executed by iOS devices, optimizing for both performance and energy consumption.

Integration with mobile-specific hardware and sensors is another key specialization area. Mobile ML frameworks extend the hardware acceleration capabilities of their cloud counterparts but with a focus on mobile-specific processors. For example, TensorFlow Lite can leverage mobile GPUs and neural processing units (NPUs) found in many modern smartphones. Qualcomm's Neural Processing SDK is designed to efficiently utilize the AI accelerators present in Snapdragon SoCs. This hardware-specific optimization often involves custom operator implementations and low-level optimizations tailored for mobile processors.

Automatic differentiation, while crucial for training in cloud environments, is often minimized or removed entirely in mobile frameworks to reduce model size and improve inference speed. Instead, mobile ML frameworks focus on efficient inference, with model updates typically performed off-device and then deployed to the mobile application.

Mobile ML frameworks also often include features for model updating and versioning, allowing for the deployment of improved models without requiring full app updates. Some frameworks support limited on-device learning, enabling models to adapt to user behavior or environmental changes without compromising data privacy.

The specializations of mobile ML frameworks collectively enable the deployment of sophisticated ML models on resource-constrained mobile devices. This expands the potential applications of AI in mobile environments, ranging from real-time image and speech recognition to personalized user experiences. However, effectively utilizing these frameworks requires careful consideration of the target device capabilities, user experience requirements, and privacy implications, necessitating a balance between model performance and resource utilization.

7.7.4 TinyML Frameworks

TinyML frameworks are specialized software infrastructures designed for deploying machine learning models on extremely resource-constrained devices, typically microcontrollers and low-power embedded systems. These frameworks address the severe limitations in processing power, memory, and energy consumption characteristic of tiny devices. The specialization of TinyML frameworks primarily focuses on extreme model compression, optimizations for severely constrained environments, and integration with microcontroller-specific architectures.

Extreme model compression in TinyML frameworks takes the quantization techniques mentioned in mobile and edge frameworks to their logical conclusion. While mobile frameworks might use 8-bit quantization, TinyML often employs even more aggressive techniques, such as 4-bit, 2-bit, or even 1-bit (binary) representations of model parameters. Frameworks like TensorFlow Lite Micro[19] exemplify this approach (David et al. 2021), pushing the boundaries of model compression to fit within the kilobytes of memory available on microcontrollers.

The execution model in TinyML frameworks is highly specialized. Unlike the dynamic graph capabilities seen in some cloud and mobile frameworks, TinyML frameworks almost exclusively use static, highly optimized graphs. The just-in-time compilation approach seen in frameworks like JAX is typically not feasible in TinyML due to memory constraints. Instead, these frameworks often employ ahead-of-time compilation techniques to generate highly optimized, device-specific code.

Memory management in TinyML frameworks is far more constrained than in other environments. While edge and mobile frameworks might use dynamic memory allocation, TinyML frameworks like uTensor often rely on static memory allocation to avoid runtime overhead and fragmentation. This approach requires careful planning of the memory layout at compile time, a stark contrast to the more flexible memory management in cloud-based frameworks.

Hardware integration in TinyML frameworks is highly specific to microcontroller architectures. Unlike the general GPU support seen in cloud frameworks or the mobile GPU/NPU support in mobile frameworks, TinyML frameworks

[19] In 2015, Google released TensorFlow which was primarily designed for the cloud. In response the need for embedded ML frameworks, they released TensorFlow Lite Micro in 2019.

often provide optimizations for specific microcontroller instruction sets. For example, ARM's CMSIS-NN (L. Lai, Suda, and Chandra 2018) provides optimized neural network kernels for Cortex-M series microcontrollers, which are often integrated into TinyML frameworks.

The concept of automatic differentiation, central to cloud-based frameworks and present to some degree in edge and mobile frameworks, is typically absent in TinyML frameworks. The focus is almost entirely on inference, with any learning or model updates usually performed off-device due to the severe computational constraints.

TinyML frameworks also specialize in power management to a degree not seen in other ML environments. Features like duty cycling and ultra-low-power wake-up capabilities are often integrated directly into the ML pipeline, enabling always-on sensing applications that can run for years on small batteries.

The extreme specialization of TinyML frameworks enables ML deployments in previously infeasible environments, from smart dust sensors to implantable medical devices. However, this specialization comes with significant trade-offs in model complexity and accuracy, requiring careful consideration of the balance between ML capabilities and the severe resource constraints of target devices.

7.8 Framework Selection

Framework selection builds on our understanding of framework specialization across computing environments. Engineers must evaluate three interdependent factors when choosing a framework: model requirements, hardware constraints, and software dependencies. The TensorFlow ecosystem demonstrates how these factors shape framework design through its variants: TensorFlow, TensorFlow Lite, and TensorFlow Lite Micro.

Table 7.4 illustrates key differences between TensorFlow variants. Each variant represents specific trade-offs between computational capability and resource requirements. These trade-offs manifest in supported operations, binary size, and integration requirements.

Table 7.4: TensorFlow framework comparison - General.

Model	TensorFlow	TensorFlow Lite	TensorFlow Lite Micro
Training	Yes	No	No
Inference	Yes *(but inefficient on edge)*	Yes *(and efficient)*	Yes *(and even* ***more*** *efficient)*
How Many Ops	~1400	~130	~50
Native Quantization Tooling + Support	No	Yes	Yes

Engineers analyze three primary aspects when selecting a framework:

1. Model requirements determine which operations and architectures the framework must support
2. Software dependencies define operating system and runtime requirements

3. Hardware constraints establish memory and processing limitations

This systematic analysis enables engineers to select frameworks that align with their deployment requirements. As we examine the TensorFlow variants, we will explore how each aspect influences framework selection and shapes the capabilities of deployed machine learning systems.

7.8.1 Model Requirements

Model architecture capabilities vary significantly across TensorFlow variants, with clear trade-offs between functionality and efficiency. Table 7.4 quantifies these differences across four key dimensions: training capability, inference efficiency, operation support, and quantization features.

TensorFlow supports approximately 1,400 operations and enables both training and inference. However, as Table 7.4 indicates, its inference capabilities are inefficient for edge deployment. TensorFlow Lite reduces the operation count to roughly 130 operations while improving inference efficiency. It eliminates training support but adds native quantization tooling. TensorFlow Lite Micro further constrains the operation set to approximately 50 operations, achieving even higher inference efficiency through these constraints. Like TensorFlow Lite, it includes native quantization support but removes training capabilities.

This progressive reduction in operations enables deployment on increasingly constrained devices. The addition of native quantization in both TensorFlow Lite and TensorFlow Lite Micro provides essential optimization capabilities absent in the full TensorFlow framework. Quantization transforms models to use lower precision operations, reducing computational and memory requirements for resource-constrained deployments.

7.8.2 Software Dependencies

Table 7.5 reveals three key software considerations that differentiate TensorFlow variants: operating system requirements, memory management capabilities, and accelerator support. These differences reflect each variant's optimization for specific deployment environments.

Table 7.5: TensorFlow framework comparison - Software.

Software	TensorFlow	TensorFlow Lite	TensorFlow Lite Micro
Needs an OS	Yes	Yes	No
Memory Mapping of Models	No	Yes	Yes
Delegation to accelerators	Yes	Yes	No

Operating system dependencies mark a fundamental distinction between variants. TensorFlow and TensorFlow Lite require an operating system, while TensorFlow Lite Micro operates without OS support. This enables TensorFlow Lite Micro to reduce memory overhead and startup time, though it can still integrate with real-time operating systems like FreeRTOS, Zephyr, and Mbed OS when needed.

Memory management capabilities also distinguish the variants. TensorFlow Lite and TensorFlow Lite Micro support model memory mapping, enabling direct model access from flash storage rather than loading into RAM. TensorFlow lacks this capability, reflecting its design for environments with abundant memory resources. Memory mapping becomes increasingly important as deployment moves toward resource-constrained devices.

Accelerator delegation capabilities further differentiate the variants. Both TensorFlow and TensorFlow Lite support delegation to accelerators, enabling efficient computation distribution. TensorFlow Lite Micro omits this feature, acknowledging the limited availability of specialized accelerators in embedded systems. This design choice maintains the framework's minimal footprint while matching typical embedded hardware configurations.

7.8.3 Hardware Constraints

Table 7.6 quantifies the hardware requirements across TensorFlow variants through three metrics: base binary size, memory footprint, and processor architecture support. These metrics demonstrate the progressive optimization for constrained computing environments.

Table 7.6: TensorFlow framework comparison: Hardware.

Hardware	TensorFlow	TensorFlow Lite	TensorFlow Lite Micro
Base Binary Size	3 MB+	100 KB	~10 KB
Base Memory Footprint	~5 MB	300 KB	20 KB
Optimized Architectures	X86, TPUs, GPUs	Arm Cortex A, x86	Arm Cortex M, DSPs, MCUs

Binary size requirements decrease significantly across variants. TensorFlow requires over 3 MB for its base binary, reflecting its comprehensive feature set. TensorFlow Lite reduces this to 100 KB by eliminating training capabilities and unused operations. TensorFlow Lite Micro achieves a remarkable 10 KB binary size through aggressive optimization and feature reduction.

Memory footprint follows a similar pattern of reduction. TensorFlow requires approximately 5 MB of base memory, while TensorFlow Lite operates within 300 KB. TensorFlow Lite Micro further reduces memory requirements to 20 KB, enabling deployment on highly constrained devices.

Processor architecture support aligns with each variant's intended deployment environment. TensorFlow supports x86 processors and accelerators including TPUs and GPUs, enabling high-performance computing in data centers. TensorFlow Lite targets mobile and edge processors, supporting Arm Cortex-A and x86 architectures. TensorFlow Lite Micro specializes in microcontroller deployment, supporting Arm Cortex-M cores, digital signal processors (DSPs), and various microcontroller units (MCUs) including STM32, NXP Kinetis, and Microchip AVR.

7.8.4 Additional Selection Factors

Framework selection for embedded systems extends beyond technical specifications of model architecture, hardware requirements, and software dependencies.

Additional factors affect development efficiency, maintenance requirements, and deployment success. These factors require systematic evaluation to ensure optimal framework selection.

7.8.4.1 Performance Optimization

Performance in embedded systems encompasses multiple metrics beyond computational speed. Framework evaluation must consider:

Inference latency determines system responsiveness and real-time processing capabilities. Memory utilization affects both static storage requirements and runtime efficiency. Power consumption impacts battery life and thermal management requirements. Frameworks must provide optimization tools for these metrics, including quantization, operator fusion, and hardware-specific acceleration.

7.8.4.2 Deployment Scalability

Scalability requirements span both technical capabilities and operational considerations. Framework support must extend across deployment scales and scenarios:

Device scaling enables consistent deployment from microcontrollers to more powerful embedded processors. Operational scaling supports the transition from development prototypes to production deployments. Version management facilitates model updates and maintenance across deployed devices. The framework must maintain consistent performance characteristics throughout these scaling dimensions.

7.9 Conclusion

AI frameworks have evolved from basic numerical libraries into sophisticated software systems that shape how we develop and deploy machine learning applications. The progression from early numerical computing to modern deep learning frameworks demonstrates the field's rapid technological advancement.

Modern frameworks like TensorFlow, PyTorch, and JAX implement distinct approaches to common challenges in machine learning development. Each framework offers varying tradeoffs between ease of use, performance, and flexibility. TensorFlow emphasizes production deployment, PyTorch focuses on research and experimentation, while JAX prioritizes functional programming patterns.

The specialization of frameworks into cloud, edge, mobile, and tiny ML implementations reflects the diverse requirements of machine learning applications. Cloud frameworks optimize for scalability and distributed computing. Edge and mobile frameworks prioritize model efficiency and reduced resource consumption. TinyML frameworks target constrained environments with minimal computing resources.

Understanding framework architecture, from tensor operations to execution models, enables developers to select appropriate tools for specific use cases, optimize application performance, debug complex computational graphs, and deploy models across different computing environments.

The continuing evolution of AI frameworks will likely focus on improving developer productivity, hardware acceleration, and deployment flexibility. These advancements will shape how machine learning systems are built and deployed across increasingly diverse computing environments.

7.10 Resources

ℹ Slides

- *Coming soon.*

❗ Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 8

AI Training

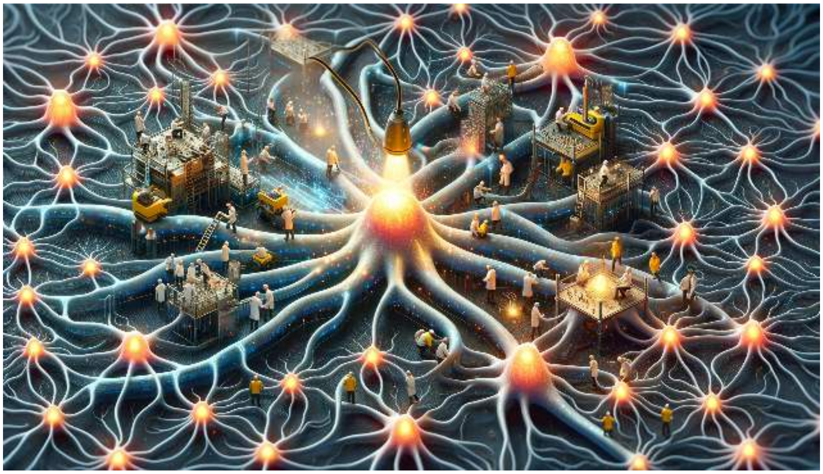


Figure 8.1: *DALL·E 3 Prompt: An illustration for AI training, depicting a neural network with neurons that are being repaired and firing. The scene includes a vast network of neurons, each glowing and firing to represent activity and learning. Among these neurons, small figures resembling engineers and scientists are actively working, repairing and tweaking the neurons. These miniature workers symbolize the process of training the network, adjusting weights and biases to achieve convergence. The entire scene is a visual metaphor for the intricate and collaborative effort involved in AI training, with the workers representing the continuous optimization and learning within a neural network. The background is a complex array of interconnected neurons, creating a sense of depth and complexity.*

Purpose

How do machine learning training workloads manifest as systems challenges, and what architectural principles guide their efficient implementation?

Machine learning training is a unique class of computational workload that demands careful orchestration of computation, memory, and data movement. The process of transforming training algorithms into efficient system implementations requires understanding how mathematical operations map to hardware resources, how data flows through memory hierarchies, and how system architectures influence training performance. Investigating these system-level considerations helps establish core principles for designing and optimizing training infrastructure. By understanding and addressing these challenges, we can develop more efficient and scalable solutions to meet the demands of modern machine learning workloads.

Learning Objectives

- Explain the link between mathematical operations and system trade-offs in AI training.
- Identify bottlenecks in training systems and their impact on performance.
- Outline the key components of training pipelines and their roles in model training.
- Determine appropriate optimization techniques to improve training efficiency.
- Analyze training systems beyond a single machine, including distributed approaches.
- Evaluate and design training processes with a focus on efficiency and scalability.

8.1 Overview

Machine learning has revolutionized modern computing by enabling systems to learn patterns from data, with training being its cornerstone. This computationally intensive process involves adjusting millions, and even billions, of parameters to minimize errors on training examples while ensuring the model generalizes effectively to unseen data. The success of machine learning models hinges on this training phase.

The training process brings together algorithms, data, and computational resources into an integrated workflow. Models, particularly deep neural networks used in domains such as computer vision and natural language processing, require significant computational effort due to their complexity and scale. Even resource-constrained models, such as those used in Mobile ML or Tiny ML applications, require careful tuning to achieve an optimal balance between accuracy, computational efficiency, and generalization.

As models have grown in size and complexity[0], the systems that enable efficient training have become increasingly sophisticated. Training systems must coordinate computation across memory hierarchies, manage data movement, and optimize resource utilization, all while maintaining numerical stability and convergence properties. This intersection of mathematical optimization with systems engineering creates unique challenges in maximizing training throughput.

This chapter examines the key components and architecture of machine learning training systems. We discuss the design of training pipelines, memory and computation systems, data management strategies, and advanced optimization techniques. Additionally, we explore distributed training frameworks and their role in scaling training processes. Real-world examples and case studies are provided to connect theoretical principles to practical implementations, offering insight into the development of efficient, scalable, and effective training systems.

[0] Model sizes have grown exponentially since AlexNet (60M parameters) in 2012, with modern large language models like GPT-4 estimated to have over 1 trillion parameters, which represents an increase of over 16,000x in just over a decade.

8.2 Training Systems

Machine learning training systems represent a distinct class of computational workload with unique demands on hardware and software infrastructure. These systems must efficiently orchestrate repeated computations over large datasets while managing substantial memory requirements and data movement patterns. Unlike traditional high-performance computing workloads, training systems exhibit specific characteristics that influence their design and implementation.

8.2.1 System Evolution

Computing system architectures have evolved through distinct generations, with each new era building upon previous advances while introducing specialized optimizations for emerging application requirements (Figure 8.2). This progression demonstrates how hardware adaptation to application needs shapes modern machine learning systems.

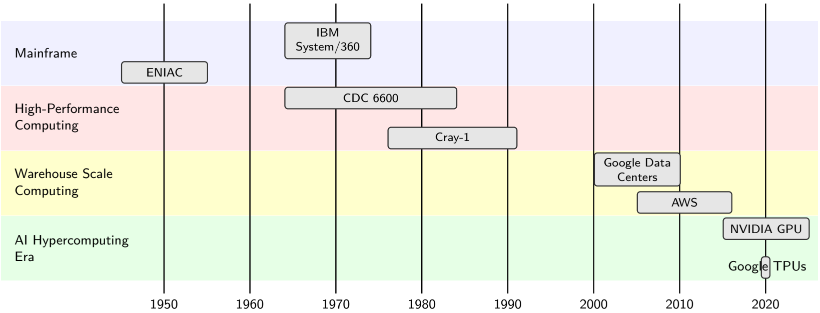


Figure 8.2: Timeline of major advancements in computing systems for machine learning, showing the evolution from mainframes to AI hypercomputing systems.

Electronic computation began with the mainframe era. ENIAC (1945) established the viability of electronic computation at scale, while the IBM System/360 (1964) introduced architectural principles of standardized instruction sets and memory hierarchies. These fundamental concepts laid the groundwork for all subsequent computing systems.

High-performance computing (HPC) systems (Thornton 1965) built upon these foundations while specializing for scientific computation. The CDC 6600 and later systems like the CM-5 (T. M. Corporation 1992) optimized for dense matrix operations and floating-point calculations.

HPC These systems implemented specific architectural features for scientific workloads: high-bandwidth memory systems for array operations, vector processing units for mathematical computations, and specialized interconnects for collective communication patterns. Scientific computing demanded emphasis on numerical precision and stability, with processors and memory systems designed for regular, predictable access patterns. The interconnects supported tightly synchronized parallel execution, enabling efficient collective operations across computing nodes.

Warehouse-scale computing marked the next evolutionary step. Google's data center implementations (Barroso and Hölzle 2007a) introduced new optimizations for internet-scale data processing. Unlike HPC systems focused on tightly coupled scientific calculations, warehouse computing handled loosely coupled tasks with irregular data access patterns.

WSC systems introduced architectural changes to support high throughput for independent tasks, with robust fault tolerance and recovery mechanisms. The storage and memory systems adapted to handle sparse data structures efficiently, moving away from the dense array optimizations of HPC. Resource management systems evolved to support multiple applications sharing the computing infrastructure, contrasting with HPC's dedicated application execution model.

Deep learning computation emerged as the next frontier, building upon this accumulated architectural knowledge. AlexNet's (Krizhevsky, Sutskever, and Hinton 2017a) success in 2012 highlighted the need for further specialization. While previous systems focused on either scientific calculations or independent data processing tasks, neural network training introduced new computational patterns. The training process required continuous updates to large sets of parameters, with complex data dependencies during model optimization. These workloads demanded new approaches to memory management and inter-device communication that neither HPC nor warehouse computing had fully addressed.

The AI hypercomputing era, beginning in 2015, represents the latest step in this evolutionary chain. NVIDIA GPUs and Google TPUs introduced hardware designs specifically optimized for neural network computations, moving beyond adaptations of existing architectures. These systems implemented new approaches to parallel processing, memory access, and device communication to handle the distinct patterns of model training. The resulting architectures balanced the numerical precision needs of scientific computing with the scale requirements of warehouse systems, while adding specialized support for the iterative nature of neural network optimization.

This architectural progression illuminates why traditional computing systems proved insufficient for neural network training. As shown in Table 8.1, while HPC systems provided the foundation for parallel numerical computation and warehouse-scale systems demonstrated distributed processing at scale, neither fully addressed the computational patterns of model training. Modern neural networks combine intensive parameter updates, complex memory access patterns, and coordinated distributed computation in ways that demanded new architectural approaches.

Table 8.1: Comparison of computing system characteristics across different eras

Era	Primary Workload	Memory Patterns	Processing Model	System Focus
Mainframe	Sequential batch processing	Simple memory hierarchy	Single instruction stream	General-purpose computation
HPC	Scientific simulation	Regular array access	Synchronized parallel	Numerical precision, collective operations
Warehouse-scale	Internet services	Sparse, irregular access	Independent parallel tasks	Throughput, fault tolerance
AI Hyper-computing	Neural network training	Parameter-heavy, mixed access	Hybrid parallel, distributed	Training optimization, model scale

Understanding these distinct characteristics and their evolution from previous computing eras explains why modern AI training systems require dedicated hardware features and optimized system designs. This historical context provides the foundation for examining machine learning training system architectures in detail.

8.2.2 System Role

The development of modern machine learning models relies critically on specialized systems for training and optimization. These systems are a complex interplay of hardware and software components that must efficiently handle massive datasets while maintaining numerical precision and computational stability. While there is no universally accepted definition of training systems due to their rapid evolution and diverse implementations, they share common characteristics and requirements that distinguish them from traditional computing infrastructures.

> Definition of Training Systems
>
> **Machine Learning Training Systems** refer to the specialized computational frameworks that manage and execute the *iterative optimization* of machine learning models. These systems encompass the *software and hardware stack* responsible for processing training data, computing gradients, updating model parameters, and coordinating distributed computation. Training systems operate at multiple scales, from single hardware accelerators to *distributed clusters*, and incorporate components for *data management*, *computation scheduling*, *memory optimization*, and *performance monitoring*. They serve as the foundational infrastructure that enables the systematic development and refinement of machine learning models through empirical training on data.

These training systems constitute the fundamental infrastructure required for developing predictive models. They execute the mathematical optimization of model parameters, converting input data into computational representations for tasks such as pattern recognition, language understanding, and decision automation. The training process involves systematic iteration over datasets to minimize error functions and achieve optimal model performance.

Training systems function as integral components within the broader machine learning pipeline. They interface with preprocessing frameworks that standardize and transform raw data, while connecting to deployment architectures that enable model serving. The computational efficiency and reliability of training systems directly influence the development cycle, from initial experimentation through model validation to production deployment.

The emergence of transformer architectures and large-scale models has introduced new requirements for training systems. Contemporary implementations must efficiently process petabyte-scale datasets, orchestrate distributed training across multiple accelerators, and optimize memory utilization for models containing billions of parameters. The management of data parallelism, model parallelism, and inter-device communication presents significant technical challenges in modern training architectures.

Training systems also significantly impact the operational considerations of machine learning development. System design must address multiple technical constraints: computational throughput, energy consumption, hardware compatibility, and scalability with increasing model complexity. These factors determine both the technical feasibility and operational viability of machine learning implementations across different scales and applications.

8.2.3 Systems Thinking

The practical execution of training models is deeply tied to system design. Training is not merely a mathematical optimization problem; it is a system-driven process that requires careful orchestration of computing hardware, memory, and data movement.

Training workflows consist of interdependent stages: data preprocessing, forward and backward passes, and parameter updates. Each stage imposes specific demands on system resources. For instance, data preprocessing relies on storage and I/O subsystems to provide computing hardware with continuous input. While traditional processors like CPUs handle many training tasks effectively, increasingly complex models have driven the adoption of hardware accelerators, such as Graphics Processing Units, GPUs, and specialized machine learning processors, that can process mathematical operations in parallel. These accelerators, alongside CPUs, handle operations like gradient computation and parameter updates. The performance of these stages depends on how well the system manages bottlenecks such as memory bandwidth and communication latency.

System constraints often dictate the performance limits of training workloads. Modern accelerators are frequently bottlenecked by memory bandwidth, as data movement between memory hierarchies can be slower and more energy-intensive than the computations themselves (D. A. Patterson and Hennessy 2021a). In distributed setups, synchronization across devices introduces additional latency, with the performance of interconnects (e.g., NVLink, InfiniBand) playing a crucial role.

Optimizing training workflows is essential to overcoming these limitations. Techniques like overlapping computation with data loading, mixed-precision training (Kuchaiev et al. 2018), and efficient memory allocation can significantly enhance performance. These optimizations ensure that accelerators are utilized effectively, minimizing idle time and maximizing throughput.

Beyond training infrastructure, systems thinking has also informed model architecture decisions. System-level constraints often guide the development of new model architectures and training approaches. For example, memory limitations have motivated research into more efficient neural network architectures (M. X. Chen et al. 2018), while communication overhead in distributed systems has influenced the design of optimization algorithms. These adaptations demonstrate how practical system considerations shape the evolution of machine learning approaches within given computational bounds.

For example, training large Transformer models requires partitioning data and model parameters across multiple devices. This introduces synchronization challenges, particularly during gradient updates. Communication libraries such as NVIDIA's Collective Communications Library (NCCL) enable efficient gradient sharing, providing the foundation for more advanced techniques we discuss in later sections. These examples illustrate how system-level considerations influence the feasibility and efficiency of modern training workflows.

8.3 Mathematical Foundations

Neural networks are grounded in mathematical principles that define their structure and functionality. These principles encompass key operations essential for enabling networks to learn complex patterns from data. A thorough understanding of the mathematical foundations underlying these operations is vital, not only for comprehending the mechanics of neural network computation but also for recognizing their broader implications at the system level.

Therefore, we need to connect the theoretical underpinnings of these operations to their practical implementation, examining how modern systems optimize these computations to address critical challenges such as memory management, computational efficiency, and scalability in training deep learning models.

8.3.1 Neural Network Computation

We have previously introduced the basic operations involved in training a neural network (see Chapter 3 and Chapter 4), such as forward propagation and the use of loss functions to evaluate performance. Here, we build on those foundational concepts to explore how these operations are executed at the system level. Key mathematical operations such as matrix multiplications and activation functions[1] underpin the system requirements for training neural networks. Foundational works by Rumelhart, Hinton, and Williams (1986) via the introduction of backpropagation and the development of efficient matrix computation libraries, e.g., BLAS (Dongarra et al. 1988), laid the groundwork for modern training architectures.

8.3.1.1 Core Operations

At the heart of a neural network is the process of forward propagation, in its simplest case, involves two primary operations: matrix multiplication and the application of an activation function. Matrix multiplication forms the basis of the linear transformation in each layer of the network. At layer l, the computation can be described as:

$$A^{(l)} = f\left(W^{(l)}A^{(l-1)} + b^{(l)}\right)$$

Where:

- $A^{(l-1)}$ represents the activations from the previous layer (or the input layer for the first layer),

[1] Activation functions are nonlinear transformations applied to neuron outputs that enable neural networks to learn complex patterns. By introducing nonlinearity between layers, they allow networks to approximate arbitrary functions. Without activation functions, neural networks would collapse into simple linear models. Like biological neurons that only fire above certain thresholds, activation functions introduce essential nonlinear behavior.

- $W^{(l)}$ is the weight matrix at layer l, which contains the parameters learned by the network,
- $b^{(l)}$ is the bias vector for layer l,
- $f(\cdot)$ is the activation function applied element-wise (e.g., ReLU, sigmoid) to introduce non-linearity.

8.3.1.2 Matrix Operations

The computational patterns in neural networks revolve around various types of matrix operations. Understanding these operations and their evolution reveals the reasons why specific system designs and optimizations emerged in machine learning training systems.

Dense Matrix-Matrix Multiplication. Matrix-matrix multiplication dominates computation in neural networks, accounting for 60-90% of training time (K. He et al. 2016b). Early neural network implementations relied on standard CPU-based linear algebra libraries. The evolution of matrix multiplication algorithms has closely followed advancements in numerical linear algebra. From Strassen's algorithm, which reduced the naive $O(n^3)$ complexity to approximately $O(n^{2.81})$ (Strassen 1969), to contemporary hardware-accelerated libraries like cuBLAS, these innovations have continually pushed the limits of computational efficiency.

Modern systems implement blocked matrix computations for parallel processing across multiple units. As neural architectures grew in scale, these multiplications began to demand significant memory resources, weight matrices and activation matrices must both remain accessible for the backward pass during training. Hardware designs adapted to optimize for these dense multiplication patterns while managing growing memory requirements.

Matrix-Vector Operations. Matrix-vector multiplication became essential with the introduction of normalization techniques in neural architectures. While computationally simpler than matrix-matrix multiplication, these operations present unique system challenges. They exhibit lower hardware utilization due to their limited parallelization potential. This characteristic influences both hardware design and model architecture decisions, particularly in networks processing sequential inputs or computing layer statistics.

Batched Operations. The introduction of batching transformed matrix computation in neural networks. By processing multiple inputs simultaneously, training systems convert matrix-vector operations into more efficient matrix-matrix operations. This approach improves hardware utilization but increases memory demands for storing intermediate results. Modern implementations must balance batch sizes against available memory, leading to specific optimizations in memory management and computation scheduling.

Hardware accelerators like Google's TPU (Jouppi, Young, et al. 2017b) reflect this evolution, incorporating specialized matrix units and memory hierarchies for these diverse multiplication patterns. These hardware adaptations enable training of large-scale models like GPT-3 (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, and al. 2020) through efficient handling of varied matrix operations.

8.3.1.3 Activation Functions

Activation functions are central to neural network operation. As shown in Figure 8.3, these functions apply different non-linear transformations to input values, which is essential for enabling neural networks to approximate complex mappings between inputs and outputs. Without activation functions, neural networks, regardless of depth, would collapse into linear systems, severely limiting their representational power (I. J. Goodfellow, Courville, and Bengio 2013a).

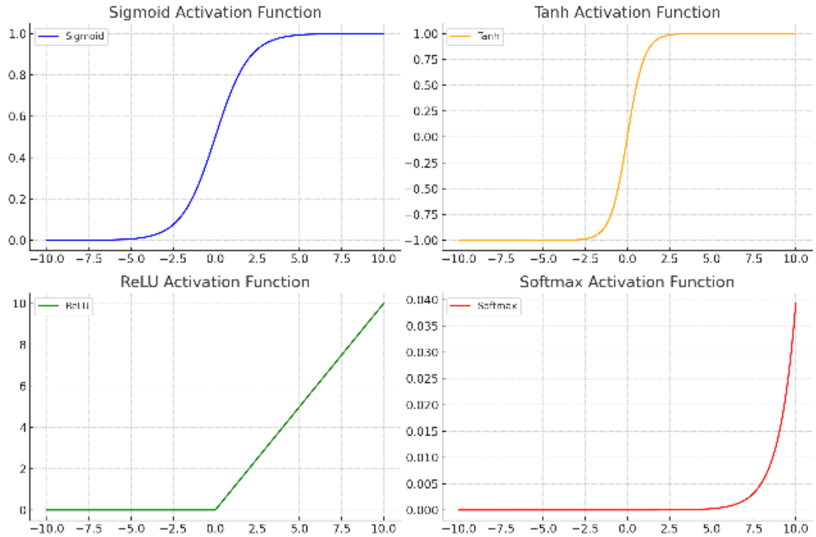


Figure 8.3: Activation functions. Note that the axes are different across graphs.

While activation functions are applied element-wise to the outputs of each neuron, their computational cost is significantly lower than that of matrix multiplications. Typically, activation functions contribute to about 5-10% of the total computation time. However, their impact on the learning process is profound, influencing not only the network's ability to learn but also its convergence rate and gradient flow.

A careful understanding of activation functions and their computational implications is vital for designing efficient machine learning pipelines. Selecting the appropriate activation function can minimize computation time without compromising the network's ability to learn complex patterns, ensuring both efficiency and accuracy.

Sigmoid. The sigmoid function is one of the original activation functions in neural networks. It maps input values to the range $(0,1)$ through the following mathematical expression:

$$\mathrm{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

This function produces an S-shaped curve, where inputs far less than zero approach an output of 0, and inputs much greater than zero approach 1. The smooth transition between these bounds makes sigmoid particularly useful in scenarios where outputs need to be interpreted as probabilities. It is therefore commonly applied in the output layer of networks for binary classification tasks.

The sigmoid function is differentiable and has a well-defined gradient, which makes it suitable for use with gradient-based optimization methods. Its bounded output ensures numerical stability, preventing excessively large activations that might destabilize the training process. However, for inputs with very high magnitudes (positive or negative), the gradient becomes negligible, which can lead to the vanishing gradient problem.[2] This issue is particularly detrimental in deep networks, where gradients must propagate through many layers during training (Hochreiter 1998).

Additionally, sigmoid outputs are not zero-centered, meaning that the function produces only positive values. This lack of symmetry can cause optimization algorithms like stochastic gradient descent (SGD)[3] to exhibit inefficient updates, as gradients may introduce biases that slow convergence. To mitigate these issues, techniques such as batch normalization[4] or careful initialization may be employed.[5]

Despite its limitations, sigmoid remains an effective choice in specific contexts. It is often used in the final layer of binary classification models, where its output can be interpreted directly as the probability of a particular class. For example, in a network designed to classify emails as either spam or not spam, the sigmoid function converts the network's raw score into a probability, making the output more interpretable.

Tanh. The hyperbolic tangent, or tanh, is a commonly used activation function in neural networks. It maps input values through a nonlinear transformation into the range $(-1, 1)$. The mathematical definition of the tanh function is:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

This function produces an S-shaped curve, similar to the sigmoid function, but with the important distinction that its output is centered around zero. Negative inputs are mapped to values in the range $[-1, 0)$, while positive inputs are mapped to values in the range $(0, 1]$. This zero-centered property makes tanh advantageous for hidden layers, as it reduces bias in weight updates and facilitates faster convergence during optimization (Yann LeCun et al. 1998).

The tanh function is smooth and differentiable, with a gradient that is well-defined for all input values. Its symmetry around zero helps balance the activations of neurons, leading to more stable and efficient learning dynamics. However, for inputs with very large magnitudes (positive or negative), the function saturates, and the gradient approaches zero. This vanishing gradient problem can impede training in deep networks.

The tanh function is often used in the hidden layers of neural networks, particularly for tasks where the input data contains both positive and negative values. Its symmetric range $(-1, 1)$ ensures balanced activations, making it well-suited for applications such as sequence modeling and time series analysis.

[2] Vanishing gradients prevent learning in deep layers as parameter updates become negligible. Conversely, exploding gradients cause rapid weight updates, leading to unstable training. Both issues can hinder convergence and degrade model performance.

[3] Stochastic Gradient Descent (SGD): Unlike full gradient descent which computes gradients over the entire dataset, SGD estimates gradients using small batches of data. This reduces memory requirements and enables frequent parameter updates, though it introduces variance in the optimization process. This variance can help escape local minima but results in less precise convergence compared to full gradient descent.

[4] Batch Normalization: A technique that normalizes the input of each layer by adjusting and scaling the activations, reducing internal covariate shift and enabling faster training.

[5] Popular deep learning frameworks like PyTorch and TensorFlow implement robust initialization schemes based on theoretical principles and empirical research. These defaults help prevent vanishing/exploding gradients and ensure stable training.

For example, tanh is widely used in recurrent neural networks (RNNs), where its bounded and symmetric properties help stabilize learning dynamics over time. While tanh has largely been replaced by ReLU in many modern architectures due to its computational inefficiencies and vanishing gradient issues, it remains a viable choice in scenarios where its range and symmetry are beneficial.

ReLU. The Rectified Linear Unit (ReLU) is one of the most widely used activation functions in modern neural networks. Its simplicity and effectiveness have made it the default choice for most machine learning architectures. The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

This function outputs the input value if it is positive and zero otherwise. Unlike sigmoid and tanh, which produce smooth, bounded outputs, ReLU introduces sparsity in the network by setting all negative inputs to zero. This sparsity can help reduce overfitting and improve computation efficiency in many scenarios.

ReLU is particularly effective in avoiding the vanishing gradient problem, as it maintains a constant gradient for positive inputs. However, it introduces another issue known as the dying ReLU problem, where neurons can become permanently inactive if they consistently output zero. This occurs when the weights cause the input to remain in the negative range. In such cases, the neuron no longer contributes to learning.

ReLU is commonly used in the hidden layers of neural networks, particularly in convolutional neural networks (CNNs) and machine learning models for image and speech recognition tasks. Its computational simplicity and ability to prevent vanishing gradients make it ideal for training deep architectures.

Softmax. The softmax function is a widely used activation function, primarily applied in the output layer of classification models. It transforms raw scores into a probability distribution, ensuring that the outputs sum to 1. This makes it particularly suitable for multi-class classification tasks, where each output represents the probability of the input belonging to a specific class.

The mathematical definition of the softmax function for a vector of inputs $\mathbf{z} = [z_1, z_2, \ldots, z_K]$ is:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, 2, \ldots, K$$

Here, K is the number of classes, z_i represents the raw score (logit) for the i-th class, and $\sigma(z_i)$ is the probability of the input belonging to that class.

Softmax has several desirable properties that make it essential for classification tasks. It converts arbitrary real-valued inputs into probabilities, with each output value in the range $(0, 1)$ and the sum of all outputs equal to 1. The function is differentiable, which allows it to be used with gradient-based optimization methods. Additionally, the probabilistic interpretation of its output is crucial for tasks where confidence levels are needed, such as object detection or language modeling.

However, softmax is sensitive to the magnitude of the input logits. Large differences in logits can lead to highly peaked distributions, where most of the probability mass is concentrated on a single class, potentially leading to overconfidence in predictions.

Softmax finds extensive application in the final layer of neural networks for multi-class classification tasks. For instance, in image classification, models such as AlexNet and ResNet employ softmax in their final layers to assign probabilities to different image categories. Similarly, in natural language processing tasks like language modeling and machine translation, softmax is applied over large vocabularies to predict the next word or token, making it an essential component in understanding and generating human language.

Trade-offs. Activation functions in neural networks significantly impact both mathematical properties and system-level performance. The selection of an activation function directly influences training time, model scalability, and hardware efficiency through three primary factors: computational cost, gradient behavior, and memory usage.

Benchmarking common activation functions on an Apple M2 single-threaded CPU reveals meaningful performance differences, as illustrated in Figure 8.4. The data demonstrates that Tanh and ReLU execute more efficiently than Sigmoid on CPU architectures, making them particularly suitable for real-time applications and large-scale systems.

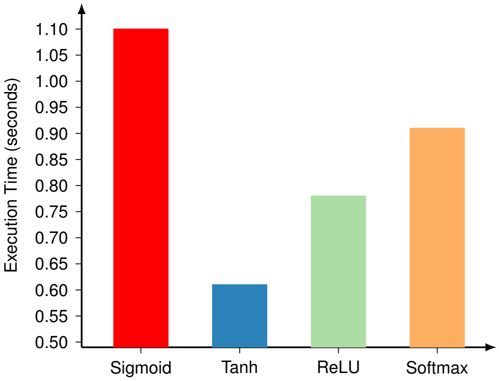


Figure 8.4: Activation function performance.

While these benchmark results provide valuable insights, they represent CPU-only performance without hardware acceleration. In production environments, modern hardware accelerators like GPUs can substantially alter the relative performance characteristics of activation functions. System architects must therefore consider their specific hardware environment and deployment context when evaluating computational efficiency.

The selection of activation functions requires careful balancing of computational considerations against mathematical properties. Key factors include the function's ability to mitigate vanishing gradients and introduce beneficial sparsity in neural activations. Each major activation function presents distinct advantages and challenges:

Sigmoid. The sigmoid function has smooth gradients and a bounded output in the range $(0,1)$, making it useful in probabilistic settings. However, the computation of the sigmoid involves an exponential function, which becomes a key consideration in both software and hardware implementations. In software, this computation is expensive and inefficient, particularly for deep networks or large datasets. Additionally, sigmoid suffers from vanishing gradients, especially for large input values, which can hinder the learning process in deep architectures. Its non-zero-centered output can also slow optimization, requiring more epochs to converge.

These computational challenges are addressed differently in hardware. Modern accelerators like GPUs and TPUs typically avoid direct computation of the exponential function, instead using lookup tables (LUTs) or piece-wise linear approximations to balance accuracy with speed. While these hardware optimizations help, the multiple memory lookups and interpolation calculations still make sigmoid more resource-intensive than simpler functions like ReLU, even on highly parallel architectures.

Tanh. The tanh function outputs values in the range $(-1,1)$, making it zero-centered and helping to stabilize gradient-based optimization algorithms. This zero-centered output helps reduce biases in weight updates, an advantage over sigmoid. Like sigmoid, however, tanh involves exponential computations that impact both software and hardware implementations. In software, this computational overhead can slow training, particularly when working with large datasets or deep models. While tanh helps prevent some of the saturation issues associated with sigmoid, it still suffers from vanishing gradients for large inputs, especially in deep networks.

In hardware, tanh leverages its mathematical relationship with sigmoid (being essentially a scaled and shifted version) to optimize implementation. Modern hardware often implement tanh using a hybrid approach: lookup tables for common input ranges combined with piece-wise approximations for edge cases. This approach helps balance accuracy with computational efficiency, though tanh remains more resource-intensive than simpler functions. Despite these challenges, tanh remains common in RNNs and LSTMs where balanced gradients are crucial.

ReLU. The ReLU function stands out for its mathematical simplicity: it passes positive values unchanged and sets negative values to zero. This straightforward behavior has profound implications for both software and hardware implementations. In software, ReLU's simple thresholding operation results in faster computation compared to sigmoid or tanh. It also helps prevent vanishing gradients and introduces beneficial sparsity in activations, as many neurons output zero. However, ReLU can suffer from the "dying ReLU" problem in deep networks, where neurons become permanently inactive and never update their weights.

The hardware implementation of ReLU showcases why it has become the dominant activation function in modern neural networks. Its simple max$(0,x)$ operation requires just a single comparison and conditional set, translating to minimal circuit complexity. Modern GPUs and TPUs can implement ReLU using a simple multiplexer that checks the input's sign bit, allowing for extremely efficient parallel processing. This hardware efficiency, combined with the sparsity it introduces, results in both reduced computation time and lower memory bandwidth requirements.

Softmax. The softmax function transforms raw logits into a probability distribution, ensuring outputs sum to 1, making it essential for classification tasks. Its computation involves exponentiating each input value and normalizing by their sum, a process that becomes increasingly complex with larger output spaces. In software, this creates significant computational overhead for tasks like natural language processing, where vocabulary sizes can reach hundreds of thousands of terms. However this is typically not a significant issue since it is often only used in the final layer. The function also requires keeping all values in memory during computation, as each output probability depends on the entire input.

At the hardware level, softmax faces unique challenges because it can't process each value independently like other activation functions. Unlike ReLU's simple threshold or even sigmoid's per-value computation, softmax needs access to all values to perform normalization. This becomes particularly demanding in modern transformer architectures, where softmax computations in attention mechanisms process thousands of values simultaneously. To manage these demands, hardware implementations often use approximation techniques or simplified versions of softmax, especially when dealing with large vocabularies or attention mechanisms.

Table 8.2 summarizes the trade-offs of these commonly used activation functions and highlights how these choices affect system performance.

Table 8.2: Comparison of different actiation functions and their advances and distagnets anad system implications.

Function	Key Advantages	Key Disadvantages	System Implications
Sigmoid	Smooth gradients; bounded output in $(0, 1)$.	Vanishing gradients; non-zero-centered output.	Exponential computation adds overhead; limited scalability for deep networks on modern accelerators.
Tanh	Zero-centered output in $(-1, 1)$; stabilizes gradients.	Vanishing gradients for large inputs.	More expensive than ReLU; effective for RNNs/LSTMs but less common in CNNs and Transformers.
ReLU	Computationally efficient; avoids vanishing gradients; introduces sparsity.	Dying neurons; unbounded output.	Simple operations optimize well on GPUs/TPUs; sparse activations reduce memory and computation needs.
Softmax	Converts logits into probabilities; sums to 1.	Computationally expensive for large outputs.	High cost for large vocabularies; hierarchical or sampled softmax needed for scalability in NLP tasks.

The choice of activation function should balance computational considerations with their mathematical properties, such as handling vanishing gradients or introducing sparsity in neural activations. This data emphasizes the importance of evaluating both theoretical and practical performance when designing neural networks. For large-scale networks or real-time applications, ReLU is often the best choice due to its efficiency and scalability. However, for tasks requiring probabilistic outputs, such as classification, softmax remains indispensable despite its computational cost. Ultimately, the ideal activation function depends on the specific task, network architecture, and hardware environment.

8.3.2 Optimization Algorithms

Optimization algorithms play an important role in neural network training by guiding the adjustment of model parameters to minimize a loss function. This process is fundamental to enabling neural networks to learn from data, and it involves finding the optimal set of parameters that yield the best model performance on a given task. Broadly, these algorithms can be divided into two categories: classical methods, which provide the theoretical foundation, and advanced methods, which introduce enhancements for improved performance and efficiency.

These algorithms are responsible for navigating the complex, high-dimensional landscape of the loss function, identifying regions where the function achieves its lowest values. This task is challenging because the loss function surface is rarely smooth or simple, often characterized by local minima, saddle points, and sharp gradients. Effective optimization algorithms are designed to overcome these challenges, ensuring convergence to a solution that generalizes well to unseen data.[6]

The selection and design of optimization algorithms have significant system-level implications, such as computation efficiency, memory requirements, and scalability to large datasets or models. A deeper understanding of these algorithms is essential for addressing the trade-offs between accuracy, speed, and resource usage.

8.3.2.1 Classical Methods

Modern neural network training relies on variations of gradient descent for parameter optimization. These approaches differ in how they process training data, leading to distinct system-level implications.

Gradient Descent. Gradient descent is the mathematical foundation of neural network training, iteratively adjusting parameters to minimize a loss function. The basic gradient descent algorithm computes the gradient of the loss with respect to each parameter, then updates parameters in the opposite direction of the gradient:

$$\theta_{t+1} = \theta_t - \alpha \nabla L(\theta_t)$$

In training systems, this mathematical operation translates into specific computational patterns. For each iteration, the system must:

1. Compute forward pass activations
2. Calculate loss value
3. Compute gradients through backpropagation
4. Update parameters using the gradient values

The computational demands of gradient descent scale with both model size and dataset size. Consider a neural network with M parameters training on N examples. Computing gradients requires storing intermediate activations during the forward pass for use in backpropagation. These activations consume

[6] When training machine learning models, a portion of data should remain completely isolated from both training and validation to provide an unbiased assessment of final model performance. This held-out test set helps evaluate how well the model generalizes to truly unseen examples.

memory proportional to the depth of the network and the number of examples being processed.

Traditional gradient descent processes the entire dataset in each iteration. For a training set with 1 million examples, computing gradients requires evaluating and storing results for each example before performing a parameter update. This approach poses significant system challenges:

$$\text{Memory Required} = N \times (\text{Activation Memory} + \text{Gradient Memory})$$

The memory requirements often exceed available hardware resources on modern hardware. A ResNet-50 model processing ImageNet-scale datasets would require hundreds of gigabytes of memory using this approach. Additionally, processing the full dataset before each update creates long iteration times, reducing the rate at which the model can learn from the data.

Stochastic Descent. These system constraints led to the development of variants that better align with hardware capabilities. The key insight was that exact gradient computation, while mathematically appealing, is not necessary for effective learning. This realization opened the door to methods that trade gradient accuracy for improved system efficiency.

These system limitations motivated the development of more efficient optimization approaches. SGD is a big shift in the optimization strategy. Rather than computing gradients over the entire dataset, SGD estimates gradients using individual training examples:

$$\theta_{t+1} = \theta_t - \alpha \nabla L(\theta_t; x_i, y_i)$$

where (x_i, y_i) represents a single training example. This approach drastically reduces memory requirements since only one example's activations and gradients need storage at any time. The stochastic nature of these updates introduces noise into the optimization process, but this noise often helps escape local minima and reach better solutions.

However, processing single examples creates new system challenges. Modern accelerators achieve peak performance through parallel computation, processing multiple data elements simultaneously. Single-example updates leave most computing resources idle, resulting in poor hardware utilization. The frequent parameter updates also increase memory bandwidth requirements, as weights must be read and written for each example rather than amortizing these operations across multiple examples.

Mini-batch Processing. Mini-batch gradient descent emerges as a practical compromise between full-batch and stochastic methods. It computes gradients over small batches of examples, enabling parallel computations that align well with modern GPU architectures (Jeffrey Dean and Ghemawat 2008).

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{B} \sum_{i=1}^{B} \nabla L(\theta_t; x_i, y_i)$$

Mini-batch processing aligns well with modern hardware capabilities. Consider a training system using GPU hardware. These devices contain thousands of cores designed for parallel computation. Mini-batch processing allows these cores to simultaneously compute gradients for multiple examples, improving hardware utilization. The batch size B becomes a key system parameter, influencing both computational efficiency and memory requirements.

The relationship between batch size and system performance follows clear patterns. Memory requirements scale linearly with batch size:

$$\text{Memory Required} = B \times (\text{Activation Memory} + \text{Gradient Memory})$$

However, larger batches enable more efficient computation through improved parallelism. This creates a trade-off between memory constraints and computational efficiency. Training systems must select batch sizes that maximize hardware utilization while fitting within available memory.

8.3.2.2 Advanced Optimization Algorithms

Advanced optimization algorithms introduce mechanisms like momentum and adaptive learning rates[7] to improve convergence. These methods have been instrumental in addressing the inefficiencies of classical approaches (Kingma and Ba 2014).

Momentum-Based Methods. Momentum methods enhance gradient descent by accumulating a velocity vector across iterations. The momentum update equations introduce an additional term to track the history of parameter updates:

$$v_{t+1} = \beta v_t + \nabla L(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

where β is the momentum coefficient, typically set between 0.9 and 0.99. From a systems perspective, momentum introduces additional memory requirements. The training system must maintain a velocity vector with the same dimensionality as the parameter vector, effectively doubling the memory needed for optimization state.

Adaptive Learning Rate Methods. RMSprop modifies the basic gradient descent update by maintaining a moving average of squared gradients for each parameter:

$$s_t = \gamma s_{t-1} + (1-\gamma)\big(\nabla L(\theta_t)\big)^2$$
$$\theta_{t+1} = \theta_t - \alpha \frac{\nabla L(\theta_t)}{\sqrt{s_t + \epsilon}}$$

This per-parameter adaptation requires storing the moving average s_t, creating memory overhead similar to momentum methods. The element-wise operations in RMSprop also introduce additional computational steps compared to basic gradient descent.

[7] Learning rate: A parameter that controls the size of parameter updates during training. A rate that is too high can cause training to diverge, while one that is too low leads to slow convergence. Finding the optimal learning rate is critical for efficient model training.

Adam Optimization. Adam combines concepts from both momentum and RMSprop, maintaining two moving averages for each parameter:

$$\begin{gathered}
m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla L(\theta_t) \\
v_t = \beta_2 v_{t-1} + (1 - \beta_2)\big(\nabla L(\theta_t)\big)^2 \\
\theta_{t+1} = \theta_t - \alpha \frac{m_t}{\sqrt{v_t + \epsilon}}
\end{gathered}$$

The system implications of Adam are more substantial than previous methods. The optimizer must store two additional vectors (m_t and v_t) for each parameter, tripling the memory required for optimization state. For a model with 100 million parameters using 32-bit floating-point numbers, the additional memory requirement is approximately 800 MB.

8.3.2.3 System Implications

The practical implementation of both classical and advanced optimization methods requires careful consideration of system resources and hardware capabilities. Understanding these implications helps inform algorithm selection and system design choices.

Trade-offs. The choice of optimization algorithm creates specific patterns of computation and memory access that influence training efficiency. Memory requirements increase progressively from basic gradient descent to more sophisticated methods:

$$\begin{gathered}
\text{Memory}_{\text{SGD}} = \text{Size}_{\text{params}} \\
\text{Memory}_{\text{Momentum}} = 2 \times \text{Size}_{\text{params}} \\
\text{Memory}_{\text{Adam}} = 3 \times \text{Size}_{\text{params}}
\end{gathered}$$

These memory costs must be balanced against convergence benefits. While Adam often requires fewer iterations to reach convergence, its per-iteration memory and computation overhead may impact training speed on memory-constrained systems.

Implementation Considerations. The efficient implementation of optimization algorithms in training frameworks hinges on strategic system-level considerations that directly influence performance. Key factors include memory bandwidth management, operation fusion techniques, and numerical precision optimization. These elements collectively determine the computational efficiency, memory utilization, and scalability of optimizers across diverse hardware architectures.

Memory bandwidth presents the primary bottleneck in optimizer implementation. Modern frameworks address this through operation fusion, which reduces memory access overhead by combining multiple operations into a single kernel. For example, the Adam optimizer's memory access requirements can grow linearly with parameter size when operations are performed separately:

$$\text{Bandwidth}_{\text{separate}} = 5 \times \text{Size}_{\text{params}}$$

However, fusing these operations into a single computational kernel significantly reduces the bandwidth requirement:

$$\text{Bandwidth}_{\text{fused}} = 2 \times \text{Size}_{\text{params}}$$

These techniques have been effectively demonstrated in systems like cuDNN and other GPU-accelerated frameworks that optimize memory bandwidth usage and operation fusion (Chetlur et al. 2014; Jouppi, Young, et al. 2017b).

Memory access patterns also play an important role in determining the efficiency of cache utilization. Sequential access to parameter and optimizer state vectors maximizes cache hit rates and effective memory bandwidth. This principle is evident in hardware such as GPUs and tensor processing units (TPUs), where optimized memory layouts significantly improve performance (Jouppi, Young, et al. 2017b).

Numerical precision represents another important tradeoff in implementation. Empirical studies have shown that optimizer states remain stable even when reduced precision formats, such as 16-bit floating-point (FP16), are used. Transitioning from 32-bit to 16-bit formats reduces memory requirements, as illustrated for the Adam optimizer:

$$\text{Memory}_{\text{Adam-FP16}} = \frac{3}{2} \times \text{Size}_{\text{params}}$$

Mixed-precision training has been shown to achieve comparable accuracy while significantly reducing memory consumption and computational overhead (Kuchaiev et al. 2018; Krishnamoorthi 2018).

The above implementation factors determine the practical performance of optimization algorithms in deep learning systems, emphasizing the importance of tailoring memory, computational, and numerical strategies to the underlying hardware architecture (T. Chen et al. 2015).

Optimizer Trade-offs. The evolution of optimization algorithms in neural network training reveals an important intersection between algorithmic efficiency and system performance. While optimizers were primarily developed to improve model convergence, their implementation significantly impacts memory usage, computational requirements, and hardware utilization.

A deeper examination of popular optimization algorithms reveals their varying impacts on system resources. As shown in Table 8.3, each optimizer presents distinct trade-offs between memory usage, computational patterns, and convergence behavior. SGD maintains minimal memory overhead, requiring storage only for model parameters and current gradients. This lightweight memory footprint comes at the cost of slower convergence and potentially poor hardware utilization due to its sequential update nature.

Table 8.3: Optimizer characteristics and system implications

Property	SGD	Momentum	RMSprop	Adam
Memory Overhead	None	Velocity terms	Squared gradients	Both velocity and squared gradients
Memory Cost	1×	2×	2×	3×

Property	SGD	Momentum	RMSprop	Adam
Access Pattern	Sequential	Sequential	Random	Random
Operations/Parameter	2	3	4	5
Hardware Efficiency	Low	Medium	High	Highest
Convergence Speed	Slowest	Medium	Fast	Fastest

Momentum methods introduce additional memory requirements by storing velocity terms for each parameter, doubling the memory footprint compared to SGD. This increased memory cost brings improved convergence through better gradient estimation, while maintaining relatively efficient memory access patterns. The sequential nature of momentum updates allows for effective hardware prefetching and cache utilization.

RMSprop adapts learning rates per parameter by tracking squared gradient statistics. Its memory overhead matches momentum methods, but its computation patterns become more irregular. The algorithm requires additional arithmetic operations for maintaining running averages and computing adaptive learning rates, increasing computational intensity from 3 to 4 operations per parameter.

Adam combines the benefits of momentum and adaptive learning rates, but at the highest system resource cost. Table 8.3 reveals that it maintains both velocity terms and squared gradient statistics, tripling the memory requirements compared to SGD. The algorithm's computational patterns involve 5 operations per parameter update, though these operations often utilize hardware more effectively due to their regular structure and potential for parallelization.

Training system designers must balance these trade-offs when selecting optimization strategies. Modern hardware architectures influence these decisions. GPUs excel at the parallel computations required by adaptive methods, while memory-constrained systems might favor simpler optimizers. The choice of optimizer affects not only training dynamics but also maximum feasible model size, achievable batch size, hardware utilization efficiency, and overall training time to convergence.

Modern training frameworks continue to evolve, developing techniques like optimizer state sharding, mixed-precision storage, and fused operations to better balance these competing demands. Understanding these system implications helps practitioners make informed decisions about optimization strategies based on their specific hardware constraints and training requirements.

8.3.3 Backpropagation Mechanics

The backpropagation algorithm computes gradients by systematically moving backward through a neural network's computational graph. While earlier discussions introduced backpropagation's mathematical principles, implementing this algorithm in training systems requires careful management of memory, computation, and data flow.

8.3.3.1 Basic Mechanics

During the forward pass, each layer in a neural network performs computations and produces activations. These activations must be stored for use during the backward pass:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = f(z^{(l)})$$

where $z^{(l)}$ represents the pre-activation values and $a^{(l)}$ represents the activations at layer l. The storage of these intermediate values creates specific memory requirements that scale with network depth and batch size.

The backward pass computes gradients by applying the chain rule, starting from the network's output and moving toward the input:

$$\frac{\partial L}{\partial z^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \odot f'(z^{(l)})$$
$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial z^{(l)}} (a^{(l-1)})^T$$

Each gradient computation requires access to stored activations from the forward pass, creating a specific pattern of memory access and computation that training systems must manage efficiently.

8.3.3.2 Backpropagation Mechanics

Neural networks learn by adjusting their parameters to reduce errors. Backpropagation computes how much each parameter contributed to the error by systematically moving backward through the network's computational graph. This process forms the computational core of the optimization algorithms discussed earlier.

For a network with parameters W_i at each layer, we need to compute $\frac{\partial L}{\partial W_i}$—how much the loss L changes when we adjust each parameter. The computation builds on the core operations covered earlier: matrix multiplications and activation functions, but in reverse order. The chain rule provides a systematic way to organize these computations:

$$\frac{\partial L_{full}}{\partial L_i} = \frac{\partial A_i}{\partial L_i} \frac{\partial L_{i+1}}{\partial A_i} \cdots \frac{\partial A_n}{\partial L_n} \frac{\partial L_{full}}{\partial A_n}$$

This equation reveals key requirements for training systems. Computing gradients for early layers requires information from all later layers, creating specific patterns in data storage and access. These patterns directly influence the efficiency of optimization algorithms like SGD or Adam discussed earlier. Modern training systems use autodifferentiation to handle these computations automatically, but the underlying system requirements remain the same.

8.3.3.3 Memory Requirements

Training systems must maintain intermediate values (activations) from the forward pass to compute gradients during the backward pass. This requirement compounds the memory demands we saw with optimization algorithms. For each layer l, the system must store:

- Input activations from the forward pass

- Output activations after applying layer operations
- Layer parameters being optimized
- Computed gradients for parameter updates

Consider a batch of training examples passing through a network. The forward pass computes and stores:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = f(z^{(l)})$$

Both $z^{(l)}$ and $a^{(l)}$ must be cached for the backward pass. This creates a multiplicative effect on memory usage: each layer's memory requirement is multiplied by the batch size, and the optimizer's memory overhead (discussed in the previous section) applies to each parameter.

The total memory needed scales with:

- Network depth (number of layers)
- Layer widths (number of parameters per layer)
- Batch size (number of examples processed together)
- Optimizer state (additional memory for algorithms like Adam)

This creates a complex set of trade-offs. Larger batch sizes enable more efficient computation and better gradient estimates for optimization, but require proportionally more memory for storing activations. More sophisticated optimizers like Adam can achieve faster convergence but require additional memory per parameter.

8.3.3.4 Memory-Computation Trade-offs

Training systems must balance memory usage against computational efficiency. Each forward pass through the network generates a set of activations that must be stored for the backward pass. For a neural network with L layers, processing a batch of B examples requires storing:

$$\text{Memory per batch} = B \times \sum_{l=1}^{L} (s_l + a_l)$$

where s_l represents the size of intermediate computations (like $z^{(l)}$) and a_l represents the activation outputs at layer l.

This memory requirement compounds with the optimizer's memory needs discussed in the previous section. The total memory consumption of a training system includes both the stored activations and the optimizer state:

$$\text{Total Memory} = \text{Memory per batch} + \text{Memory}_{\text{optimizer}}$$

To manage these substantial memory requirements, training systems use several sophisticated strategies. Gradient checkpointing is a basic approach, strategically recomputing some intermediate values during the backward pass rather than storing them. While this increases computational work, it can significantly reduce memory usage, enabling training of deeper networks or larger batch sizes on memory-constrained hardware (T. Chen et al. 2016).

The efficiency of these memory management strategies depends heavily on the underlying hardware architecture. GPU systems, with their high computational throughput but limited memory bandwidth, often encounter different bottlenecks than CPU systems. Memory bandwidth limitations on GPUs mean that even when sufficient storage exists, moving data between memory and compute units can become the primary performance constraint (Jouppi, Young, et al. 2017b).

These hardware considerations guide the implementation of backpropagation in modern training systems. Specialized memory-efficient algorithms for operations like convolutions compute gradients in tiles or chunks, adapting to available memory bandwidth. Dynamic memory management tracks the lifetime of intermediate values throughout the computation graph, deallocating memory as soon as tensors become unnecessary for subsequent computations (Paszke et al. 2019).

8.3.4 System Implications

Efficiently managing the forward pass, backward pass, and parameter updates requires a holistic understanding of how these operations interact with data loading, preprocessing pipelines, and hardware accelerators. For instance, matrix multiplications shape decisions about batch size, data parallelism, and memory allocation, while activation functions influence convergence rates and require careful trade-offs between computational efficiency and learning dynamics.

These operations set the stage for addressing the challenges of training pipeline architecture. From designing workflows for data preprocessing to employing advanced techniques like mixed-precision training, gradient accumulation, and checkpointing, their implications are far-reaching.

8.4 Pipeline Architecture

A training pipeline is the framework that governs how raw data is transformed into a trained machine learning model. Within the confines of a single system, it orchestrates the steps necessary for data preparation, computational execution, and model evaluation. The design of such pipelines is critical to ensure that training is both efficient and reproducible, allowing machine learning workflows to operate reliably.

As shown in Figure 8.5, the training pipeline consists of three main components: the data pipeline for ingestion and preprocessing, the training loop that handles model updates, and the evaluation pipeline for assessing performance. These components work together in a coordinated manner, with processed batches flowing from the data pipeline to the training loop, and evaluation metrics providing feedback to guide the training process.

8.4.1 Architectural Overview

The architecture of a training pipeline is organized around three interconnected components: the data pipeline, the training loop, and the evaluation pipeline.

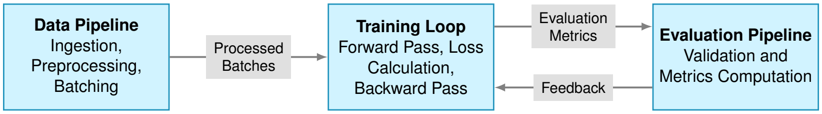

Data Pipeline
Ingestion, Preprocessing, Batching
Processed Batches
Training Loop
Forward Pass, Loss Calculation, Backward Pass
Evaluation Metrics
Feedback
Evaluation Pipeline
Validation and Metrics Computation

Figure 8.5: Training pipeline showing the three main components. The arrows indicate the flow of data and feedback between components.

These components collectively process raw data, train the model, and assess its performance, ensuring that the training process is efficient and effective.

The data pipeline initiates the process by ingesting raw data and transforming it into a format suitable for the model. This data is passed to the training loop, where the model performs its core computations to learn from the inputs. Periodically, the evaluation pipeline assesses the model's performance using a separate validation dataset. This modular structure ensures that each stage operates efficiently while contributing to the overall workflow.

8.4.1.1 Data Pipeline

The data pipeline manages the ingestion, preprocessing, and batching of data for training. Raw data is typically loaded from local storage and transformed dynamically during training to avoid redundancy and enhance diversity. For instance, image datasets may undergo preprocessing steps like normalization, resizing, and augmentation to improve the robustness of the model. These operations are performed in real time to minimize storage overhead and adapt to the specific requirements of the task (Yann LeCun et al. 1998). Once processed, the data is packaged into batches and handed off to the training loop.

8.4.1.2 Training Loop

The training loop is the computational core of the pipeline, where the model learns from the input data. Figure 8.6 illustrates this process, highlighting the forward pass, loss computation, and parameter updates on a single GPU:

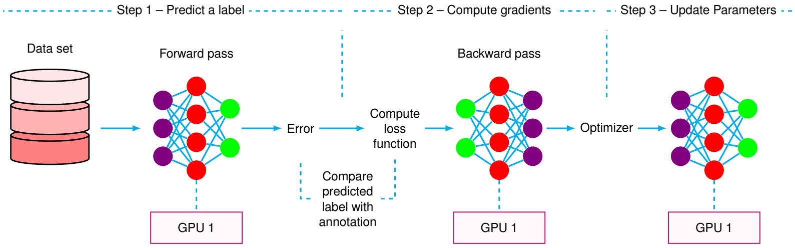


Figure 8.6: Training loop on a single GPU system.

Each iteration of the training loop involves several key steps:

1. **Step 1 – Forward Pass**: A batch of data from the dataset is passed through the neural network on the GPU to generate predictions. The model applies matrix multiplications and activation functions to transform the input into meaningful outputs.
2. **Step 2 – Compute Gradients**: The predicted values are compared with the ground truth labels to compute the error using a loss function. The loss function outputs a scalar value that quantifies the model's performance.

This error signal is then propagated backward through the network using backpropagation, which applies the chain rule of differentiation to compute gradients for each layer's parameters. These gradients indicate the necessary adjustments required to minimize the loss.

3. **Step 3 – Update Parameters**: The computed gradients are passed to an optimizer, which updates the model's parameters to minimize the loss. Different optimization algorithms, such as SGD or Adam, influence how the parameters are adjusted. The choice of optimizer impacts convergence speed and stability.

This process repeats iteratively across multiple batches and epochs, gradually refining the model to improve its predictive accuracy.

8.4.1.3 Evaluation Pipeline

The evaluation pipeline provides periodic feedback on the model's performance during training. Using a separate validation dataset, the model's predictions are compared against known outcomes to compute metrics such as accuracy or loss. These metrics help to monitor progress and detect issues like overfitting or underfitting. Evaluation is typically performed at regular intervals, such as at the end of each epoch, ensuring that the training process aligns with the desired objectives.

8.4.1.4 Component Integration

The data pipeline, training loop, and evaluation pipeline are tightly integrated to ensure a smooth and efficient workflow. Data preparation often overlaps with computation, such as when preprocessing the next batch while the current batch is being processed in the training loop. Similarly, the evaluation pipeline operates in tandem with training, providing insights that inform adjustments to the model or training procedure. This integration minimizes idle time for the system's resources and ensures that training proceeds without interruptions.

8.4.2 Data Pipeline

The data pipeline moves data from storage to computational devices during training. Like a highway system moving vehicles from neighborhoods to city centers, the data pipeline transports training data through multiple stages to reach computational resources.

The data pipeline running on the CPU serves as a bridge between raw data storage and GPU computation. As shown in Figure 8.7, the pipeline consists of three main zones: storage, CPU preprocessing, and GPU training. Each zone plays a distinct role in preparing and delivering data for model training.

In the storage zone, raw data resides on disk, typically in formats like image files for computer vision tasks or text files for natural language processing. The CPU preprocessing zone handles the transformation of this raw data through multiple stages. For example, in an image recognition model, these stages include:

1. Format conversion: Reading image files and converting them to standardized formats

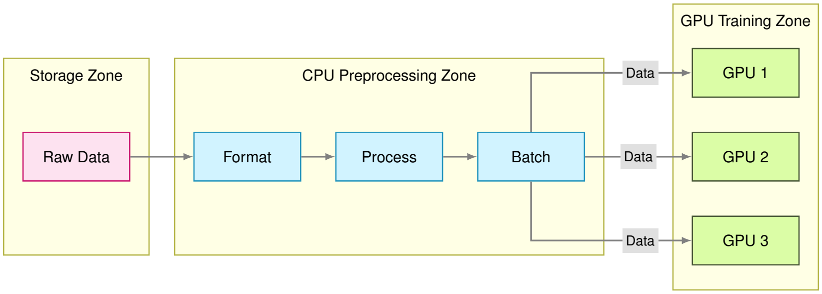


Figure 8.7: Data pipeline architecture illustrating the flow of data from raw storage through CPU preprocessing stages to GPU training units.

2. Processing: Applying operations like resizing, normalization, and data augmentation
3. Batching: Organizing processed examples into batches for efficient GPU computation

The final zone shows multiple GPUs receiving preprocessed batches for training. This organization ensures that each GPU maintains a steady supply of data, maximizing computational efficiency and minimizing idle time. The effectiveness of this pipeline directly impacts training performance, as any bottleneck in data preparation can leave expensive GPU resources underutilized.

8.4.2.1 Core Components

The performance of machine learning systems is fundamentally constrained by storage access speed, which determines the rate at which training data can be retrieved. This access speed is governed by two primary hardware constraints: disk bandwidth and network bandwidth. The maximum theoretical throughput is determined by the following relationship:

$$T_{\text{storage}} = \min(B_{\text{disk}}, B_{\text{network}})$$

where B_{disk} is the physical disk bandwidth (the rate at which data can be read from storage devices) and B_{network} represents the network bandwidth (the rate of data transfer across distributed storage systems). Both quantities are measured in bytes per second.

However, the actual throughput achieved during training operations typically falls below this theoretical maximum due to non-sequential data access patterns. The effective throughput can be expressed as:

$$T_{\text{effective}} = T_{\text{storage}} \times F_{\text{access}}$$

where F_{access} represents the access pattern factor. In typical training scenarios, F_{access} approximates 0.1, indicating that effective throughput achieves only 10% of the theoretical maximum. This significant reduction occurs because storage systems are optimized for sequential access patterns rather than the random access patterns common in training procedures.

This relationship between theoretical and effective throughput has important implications for system design and training optimization. Understanding these constraints allows practitioners to make informed decisions about data pipeline architecture and training methodology.

8.4.2.2 Preprocessing

As the data becomes available, data preprocessing transforms raw input data into a format suitable for model training. This process, traditionally implemented through Extract-Transform-Load (ETL) or Extract-Load-Transform (ELT) pipelines, is a critical determinant of training system performance. The throughput of preprocessing operations can be expressed mathematically as:

$$T_{\text{preprocessing}} = \frac{N_{\text{workers}}}{t_{\text{transform}}}$$

This equation captures two key factors:

- N_{workers} represents the number of parallel processing threads
- $t_{\text{transform}}$ represents the time required for each transformation operation

Modern training architectures employ multiple processing threads to ensure preprocessing keeps pace with the consumption rates. This parallel processing approach is essential for maintaining efficient high processor utilization.

The final stage of preprocessing involves transferring the processed data to computational devices (typically GPUs). The overall training throughput is constrained by three factors, expressed as:

$$T_{\text{training}} = \min(T_{\text{preprocessing}}, B_{\text{GPU\_transfer}}, B_{\text{GPU\_compute}})$$

where:

- $B_{\text{GPU\_transfer}}$ represents GPU memory bandwidth
- $B_{\text{GPU\_compute}}$ represents GPU computational throughput

This relationship illustrates a fundamental principle in training system design: the system's overall performance is limited by its slowest component. Whether preprocessing speed, data transfer rates, or computational capacity, the bottleneck stage determines the effective training throughput of the entire system. Understanding these relationships enables system architects to design balanced training pipelines where preprocessing capacity aligns with computational resources, ensuring optimal resource utilization.

8.4.2.3 System Implications

The relationship between data pipeline architecture and computational resources fundamentally determines the performance of machine learning training systems. This relationship can be simply expressed through a basic throughput equation:

$$T_{\text{system}} = \min(T_{\text{pipeline}}, T_{\text{compute}})$$

where T_{system} represents the overall system throughput, constrained by both pipeline throughput (T_{pipeline}) and computational speed (T_{compute}).

To illustrate these constraints, consider image classification systems. The performance dynamics can be analyzed through two critical metrics. The GPU Processing Rate (R_{GPU}) represents the maximum number of images a GPU can process per second, determined by model architecture complexity and GPU hardware capabilities. The Pipeline Delivery Rate (R_{pipeline}) is the rate at which the data pipeline can deliver preprocessed images to the GPU.

In this case, at a high level, the system's effective training speed is governed by the lower of these two rates. When R_{pipeline} is less than R_{GPU}, the system experiences underutilization of GPU resources. The degree of GPU utilization can be expressed as:

$$\text{GPU Utilization} = \frac{R_{\text{pipeline}}}{R_{\text{GPU}}} \times 100\%$$

Let us consider an example. A ResNet-50 model implemented on modern GPU hardware might achieve a processing rate of 1000 images per second. However, if the data pipeline can only deliver 200 images per second, the GPU utilization would be merely 20%, meaning the GPU remains idle 80% of the time. This results in significantly reduced training efficiency. Importantly, this inefficiency persists even with more powerful GPU hardware, as the pipeline throughput becomes the limiting factor in system performance. This demonstrates why balanced system design, where pipeline and computational capabilities are well-matched, is crucial for optimal training performance.

8.4.2.4 Data Flows

Machine learning systems manage complex data flows through multiple memory tiers while coordinating pipeline operations. The interplay between memory bandwidth constraints and pipeline execution directly impacts training performance. The maximum data transfer rate through the memory hierarchy is bounded by:

$$T_{\text{memory}} = \min(B_{\text{storage}}, B_{\text{system}}, B_{\text{accelerator}})$$

Where bandwidth varies significantly across tiers:

- Storage (B_{storage}): NVMe storage devices provide 1-2 GB/s
- System (B_{system}): Main memory transfers data at 50-100 GB/s
- Accelerator ($B_{\text{accelerator}}$): GPU memory achieves 900 GB/s or higher

These order-of-magnitude differences create distinct performance characteristics that must be carefully managed. The total time required for each training iteration comprises multiple pipelined operations:

$$t_{\text{iteration}} = \max(t_{\text{fetch}}, t_{\text{process}}, t_{\text{transfer}})$$

This equation captures three components: storage read time ($t_{\text{fetch?}}$), preprocessing time (t_{process}), and accelerator transfer time (t_{transfer}).

Modern training architectures optimize performance by overlapping these operations. When one batch undergoes preprocessing, the system simultaneously fetches the next batch from storage while transferring the previously processed batch to accelerator memory.

This coordinated movement requires precise management of system resources, particularly memory buffers and processing units. The memory hierarchy must account for bandwidth disparities while maintaining continuous data flow. Effective pipelining minimizes idle time and maximizes resource utilization through careful buffer sizing and memory allocation strategies. The successful orchestration of these components enables efficient training across the memory hierarchy while managing the inherent bandwidth constraints of each tier.

8.4.2.5 Practical Architectures

The ImageNet dataset serves as a canonical example for understanding data pipeline requirements in modern machine learning systems. This analysis examines system performance characteristics when training vision models on large-scale image datasets.

Storage performance in practical systems follows a defined relationship between theoretical and practical throughput:

$$T_{\text{practical}} = 0.5 \times B_{\text{theoretical}}$$

To illustrate this relationship, consider an NVMe storage device with 3GB/s theoretical bandwidth. Such a device achieves approximately 1.5GB/s sustained read performance. However, the random access patterns required for training data shuffling further reduce this effective bandwidth by 90%. System designers must account for this reduction through careful memory buffer design.

The total memory requirements for the system scale with batch size according to the following relationship:

$$M_{\text{required}} = (B_{\text{prefetch}} + B_{\text{processing}} + B_{\text{transfer}}) \times S_{\text{batch}}$$

In this equation, B_{prefetch} represents memory allocated for data prefetching, $B_{\text{processing}}$ represents memory required for active preprocessing operations, B_{transfer} represents memory allocated for accelerator transfers, and S_{batch} represents the training batch size.

Preprocessing operations introduce additional computational requirements. Common operations such as image resizing, augmentation, and normalization consume CPU resources. These preprocessing operations must satisfy a fundamental time constraint:

$$t_{\text{preprocessing}} < t_{\text{GPU\_compute}}$$

This inequality plays a crucial role in determining system efficiency. When preprocessing time exceeds GPU computation time, accelerator utilization decreases proportionally. The relationship between preprocessing and computation time thus establishes fundamental efficiency limits in training system design.

8.4.3 Forward Pass

The forward pass is the phase where input data propagates through the model, layer by layer, to generate predictions. Each layer performs mathematical operations such as matrix multiplications and activations, progressively transforming the data into meaningful outputs. While the conceptual flow of the forward pass is straightforward, it poses several system-level challenges that are critical for efficient execution.

8.4.3.1 Compute Operations

The forward pass in deep neural networks orchestrates a diverse set of computational patterns, each optimized for specific neural network operations. Understanding these patterns and their efficient implementation is fundamental to machine learning system design.

At their core, neural networks rely heavily on matrix multiplications, particularly in fully connected layers. The basic transformation follows the form:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$$

Here, $W^{(l)}$ represents the weight matrix, $a^{(l-1)}$ contains activations from the previous layer, and $b^{(l)}$ is the bias vector. For a layer with N neurons in the current layer and M neurons in the previous layer, processing a batch of B samples requires $N \times M \times B$ floating-point operations. A typical layer with dimensions of 512×1024 processing a batch of 64 samples executes over 33 million operations.

Modern neural architectures extend beyond these basic matrix operations to include specialized computational patterns. Convolutional networks, for instance, perform systematic kernel operations across input tensors. Consider a typical input tensor of dimensions $64 \times 224 \times 224 \times 3$ (batch size $\times$ height $\times$ width $\times$ channels) processed by 7×7 kernels. Each position requires 147 multiply-accumulate operations, and with 64 filters operating across 218×218 spatial dimensions, the computational demands become substantial.

Transformer architectures introduce attention mechanisms, which compute similarity scores between sequences. These operations combine matrix multiplications with softmax normalization, requiring efficient broadcasting and reduction operations across varying sequence lengths. The computational pattern here differs significantly from convolutions, demanding flexible execution strategies from hardware accelerators.

Throughout these networks, element-wise operations play a crucial supporting role. Activation functions like ReLU and sigmoid transform values independently. While conceptually simple, these operations can become bottlenecked by memory bandwidth rather than computational capacity, as they perform relatively few calculations per memory access. Batch normalization presents similar challenges, computing statistics and normalizing values across batch dimensions while creating synchronization points in the computation pipeline.

Modern hardware accelerators, particularly GPUs, optimize these diverse computations through massive parallelization. However, achieving peak performance requires careful attention to hardware architecture. GPUs process data in fixed-size blocks of threads called warps (in NVIDIA architectures) or wavefronts (in AMD architectures). Peak efficiency occurs when matrix dimensions align with these hardware-specific sizes. For instance, NVIDIA GPUs typically achieve optimal performance when processing matrices aligned to 32×32 dimensions.

Libraries like cuDNN address these challenges by providing optimized implementations for each operation type. These systems dynamically select algorithms based on input dimensions, hardware capabilities, and memory constraints. The selection process balances computational efficiency with memory usage, often requiring empirical measurement to determine optimal configurations for specific hardware setups.

The relationship between batch size and hardware utilization illuminates these trade-offs. When batch size decreases from 32 to 16, GPU utilization often drops due to incomplete warp occupation. While larger batch sizes improve hardware utilization, memory constraints in modern architectures may necessitate smaller batches, creating a fundamental tension between computational efficiency and memory usage. This balance exemplifies a central challenge in machine learning systems: maximizing computational throughput within hardware resource constraints.

8.4.3.2 Memory Management

Memory management is a critical challenge in general, but it is particularly crucial during the forward pass when intermediate activations must be stored for subsequent backward propagation. The total memory footprint grows with both network depth and batch size, following a basic relationship.

$$\text{Total Memory} \sim B \times \sum_{l=1}^{L} A_l$$

where B represents the batch size, L is the number of layers, and A_l represents the activation size at layer l. This simple equation masks considerable complexity in practice.

Consider ResNet-50 processing images at 224×224 resolution with a batch size of 32. The initial convolutional layer produces activation maps of dimension $112 \times 112 \times 64$. Using single-precision floating-point format (4 bytes per value), this single layer's activation storage requires approximately 98 MB. As the network progresses through its 50 layers, the dimensions of these activation maps change, typically decreasing in spatial dimensions while increasing in channel depth, creating a cumulative memory demand that can reach several gigabytes.

Modern GPUs typically provide between 16 and 24 GB of memory, which must accommodate not just these activations but also model parameters, gradients, and optimization states. This constraint has motivated several memory management strategies:

Activation checkpointing trades computational cost for memory efficiency by strategically discarding and recomputing activations during the backward pass. Rather than storing all intermediate values, the system maintains checkpoints at selected layers. During backpropagation, it regenerates necessary activations from these checkpoints. While this approach can reduce memory usage by 50% or more, it typically increases computation time by 20-30%.

Mixed precision training offers another approach to memory efficiency. By storing activations in half-precision (FP16) format instead of single-precision (FP32), memory requirements are immediately halved. Modern hardware architectures provide specialized support for these reduced-precision operations, often maintaining computational throughput while saving memory.

The relationship between batch size and memory usage creates practical trade-offs in training regimes. While larger batch sizes can improve computational efficiency, they proportionally increase memory demands. A machine learning practitioner might start with large batch sizes during initial development on smaller networks, then adjust downward when scaling to deeper architectures or when working with memory-constrained hardware.

This memory management challenge becomes particularly acute in state-of-the-art models. Recent transformer architectures can require tens of gigabytes just for activations, necessitating sophisticated memory management strategies or distributed training approaches. Understanding these memory constraints and management strategies proves essential for designing and deploying machine learning systems effectively.

8.4.4 Backward Pass

8.4.4.1 Compute Operations

The backward pass involves processing parameter gradients in reverse order through the network's layers. Computing these gradients requires matrix operations that demand significant memory and processing power.

Neural networks store activation values from each layer during the forward pass. Computing gradients combines these stored activations with gradient signals to generate weight updates. This design requires twice the memory compared to forward computation. Consider the gradient computation for a layer's weights:

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} \cdot \left(a^{(l-1)}\right)^T$$

The gradient signals $\delta^{(l)}$ at layer l multiply with transposed activations $a^{(l-1)}$ from layer $l-1$. This matrix multiplication forms the primary computational load. For example, in a layer with 1000 input features and 100 output features, computing gradients requires multiplying matrices of size $100 \times$ batch_size and batch_size $\times$ 1000, resulting in millions of floating-point operations.

8.4.4.2 Memory Operations

The backward pass moves large amounts of data between memory and compute units. Each time a layer computes gradients, it orchestrates a sequence of memory operations. The GPU first loads stored activations from memory, then reads incoming gradient signals, and finally writes the computed gradients back to memory.

To understand the scale of these memory transfers, consider a convolutional layer processing a batch of 64 images. Each image measures 224×224 pixels with 3 color channels. The activation maps alone occupy 0.38 GB of memory, storing 64 copies of the input images. The gradient signals expand this memory usage significantly - they require 8.1 GB to hold gradients for each of the layer's 64 filters. Even the weight gradients, which only store updates for the convolutional kernels, need 0.037 GB[8].

Moreover, the backward pass in neural networks require coordinated data movement through a hierarchical memory system. During backpropagation, each computation requires specific activation values from the forward pass,

[8] Memory calculations:– Activation maps:64 × 224 × 224 × 3 × 4 bytes = 0.38 GB– Gradient signals:64 × 224 × 224 × 64 × 4 bytes = 8.1 GB– Weight gradients:7 × 7 × 3 × 64 × 4 bytes = 0.037 GB

creating a pattern of data movement between memory levels. This movement pattern shapes the performance characteristics of neural network training.

These backward pass computations operate across a memory hierarchy that balances speed and capacity requirements. When computing gradients, the processor must retrieve activation values stored in high-bandwidth memory (HBM) or system memory, transfer them to fast static RAM (SRAM) for computation, and write results back to larger storage. Each gradient calculation triggers this sequence of memory transfers, making memory access patterns a key factor in backward pass performance. The frequent transitions between memory levels introduce latency that accumulates across the backward pass computation chain.

8.4.4.3 Real-World Considerations

Consider training a ResNet-50 model on the ImageNet dataset with a batch of 64 images. The first convolutional layer applies 64 filters of size 7×7 to RGB images sized 224×224. During the backward pass, this single layer's computation requires:

$$\text{Memory per image} = 224 \times 224 \times 64 \times 4 \text{ bytes}$$

The total memory requirement multiplies by the batch size of 64, reaching approximately 3.2 GB just for storing gradients. When we add memory for activations, weight updates, and intermediate computations, a single layer approaches the memory limits of many GPUs.

Deeper in the network, layers with more filters demand even greater resources. A mid-network convolutional layer might use 256 filters, quadrupling the memory and computation requirements. The backward pass must manage these resources while maintaining efficient computation. Each layer's computation can only begin after receiving gradient signals from the subsequent layer, creating a strict sequential dependency in memory usage and computation patterns.

This dependency means the GPU must maintain a large working set of memory throughout the backward pass. As gradients flow backward through the network, each layer temporarily requires peak memory usage during its computation phase. The system cannot release this memory until the layer completes its gradient calculations and passes the results to the previous layer.

8.4.5 Parameter Updates and Optimizers

The process of updating model parameters is a fundamental operation in machine learning systems. During training, after gradients are computed in the backward pass, the system must allocate and manage memory for both the parameters and their gradients, then perform the update computations. The choice of optimizer determines not only the mathematical update rule, but also the system resources required for training.

Listing 8.1 shows the parameter update process in a machine learning framework.

These operations initiate a sequence of memory accesses and computations. The system must load parameters from memory, compute updates using the

Listing 8.1: Parameter update step using backward and optimizer

```
loss.backward()  # Compute gradients
optimizer.step() # Update parameters
```

stored gradients, and write the modified parameters back to memory. Different optimizers vary in their memory requirements and computational patterns, directly affecting system performance and resource utilization.

8.4.5.1 Memory Requirements

Gradient descent, the most basic optimization algorithm that we discussed earlier, illustrates the fundamental memory and computation patterns in parameter updates. From a systems perspective, each parameter update must:

1. Read the current parameter value from memory
2. Access the computed gradient from memory
3. Perform the multiplication and subtraction operations
4. Write the new parameter value back to memory

Because gradient descent only requires memory for storing parameters and gradients, it has relatively low memory overhead compared to more complex optimizers. However, more advanced optimizers introduce additional memory requirements and computational complexity. For example, as we discussed previously, Adam maintains two extra vectors for each parameter: one for the first moment (the moving average of gradients) and one for the second moment (the moving average of squared gradients). This triples the memory usage but can lead to faster convergence. Consider the situation where there are 100,000 parameters, and each gradient requires 4 bytes (32 bits):

- Gradient Descent: 100,000 × 4 bytes = 400,000 bytes = 0.4 MB
- Adam: 3 × 100,000 × 4 bytes = 1,200,000 bytes = 1.2 MB

This problem becomes especially apparent for billion parameter models, as model sizes (without counting optimizer states and gradients) alone can already take up significant portions of GPU memory. As one way of solving this problem, the authors of GaLoRE tackle this by compressing optimizer state and gradients and computing updates in this compressed space (J. Zhao et al. 2024), greatly reducing memory footprint as shown below in Figure 8.8.

8.4.5.2 Computational Load

The computational cost of parameter updates also depends on the optimizer's complexity. For gradient descent, each update involves simple gradient calculation and application. More sophisticated optimizers like Adam require additional calculations, such as computing running averages of gradients and their squares. This increases the computational load per parameter update.

The efficiency of these computations on modern hardware like GPUs and TPUs depends on how well the optimizer's operations can be parallelized. While

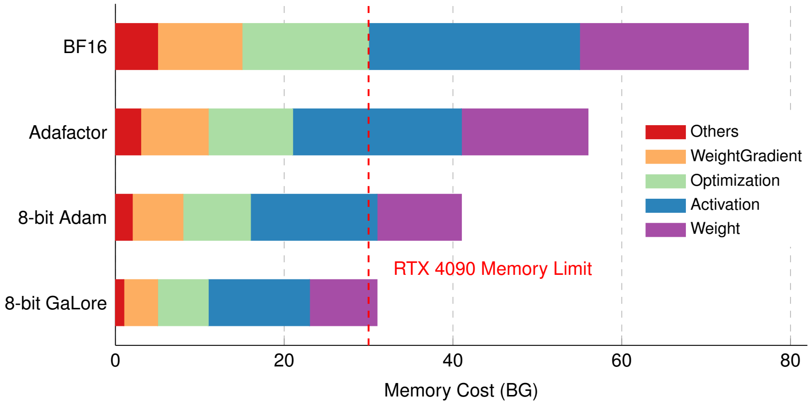


Figure 8.8: Example memory footprint breakdown for the Llama-7B model under different optimized training schemes. Note that in the unoptimized bfloat16 case, how optimizer state and weight gradients combined can take up more than double the footprint of the model weights.

matrix operations in Adam may be efficiently handled by these accelerators, some operations in complex optimizers might not parallelize well, potentially leading to hardware underutilization.

In summary, the choice of optimizer directly impacts both system memory requirements and computational load. More sophisticated optimizers often trade increased memory usage and computational complexity for potentially faster convergence, presenting important considerations for system design and resource allocation in ML systems.

8.4.5.3 Batch Size and Parameter Updates

Batch size, a critical hyperparameter in machine learning systems, significantly influences the parameter update process, memory usage, and hardware efficiency. It determines the number of training examples processed in a single iteration before the model parameters are updated.

Larger batch sizes generally provide more accurate gradient estimates, potentially leading to faster convergence and more stable parameter updates. However, they also increase memory demands proportionally:

$$\text{Memory for Batch} = \text{Batch Size} \times \text{Size of One Training Example}$$

This increase in memory usage directly affects the parameter update process, as it determines how much data is available for computing gradients in each iteration.

Larger batches tend to improve hardware utilization, particularly on GPUs and TPUs optimized for parallel processing. This can lead to more efficient parameter updates and faster training times, provided sufficient memory is available.

However, there's a trade-off to consider. While larger batches can improve computational efficiency by allowing more parallel computations during gradient calculation and parameter updates, they also require more memory. On systems with limited memory, this might necessitate reducing the batch size, potentially slowing down training or leading to less stable parameter updates.

The choice of batch size interacts with various aspects of the optimization process. For instance, it affects the frequency of parameter updates: larger batches result in less frequent but potentially more impactful updates. Additionally, batch size influences the behavior of adaptive optimization algorithms, which may need to be tuned differently depending on the batch size. In distributed training, which we discuss later, batch size often determines the degree of data parallelism, impacting how gradient computations and parameter updates are distributed across devices.

Determining the optimal batch size involves balancing these factors within hardware constraints. It often requires experimentation to find the sweet spot that maximizes both learning efficiency and hardware utilization while ensuring effective parameter updates.

8.5 Pipeline Optimizations

Efficient training of machine learning models is constrained by bottlenecks in data transfer, computation, and memory usage. These limitations manifest in specific ways: data transfer delays occur when loading training batches from disk to GPU memory, computational bottlenecks arise during matrix operations in forward and backward passes, and memory constraints emerge when storing large intermediate values like activation maps.

These bottlenecks often lead to underutilized hardware, prolonged training times, and restricted model scalability. For machine learning practitioners, understanding and implementing pipeline optimizations enables training of larger models, faster experimentation cycles, and more efficient use of available computing resources.

Here, we explore three widely adopted optimization strategies that address key performance bottlenecks in training pipelines:

1. **Prefetching and Overlapping**: Techniques to minimize data transfer delays and maximize GPU utilization.
2. **Mixed-Precision Training**: A method to reduce memory demands and computational load using lower precision formats.
3. **Gradient Accumulation and Checkpointing**: Strategies to overcome memory limitations during backpropagation and parameter updates.

Each technique is discussed in detail, covering its mechanics, advantages, and practical considerations.

8.5.1 Prefetching and Overlapping

Training machine learning models involves significant data movement between storage, memory, and computational units. The data pipeline consists of sequential transfers: from disk storage to CPU memory, CPU memory to GPU memory, and through the GPU processing units. In standard implementations, each transfer must complete before the next begins, as shown in Figure 8.9, resulting in computational inefficiencies.

Prefetching addresses these inefficiencies by loading data into memory before its scheduled computation time. During the processing of the current batch,

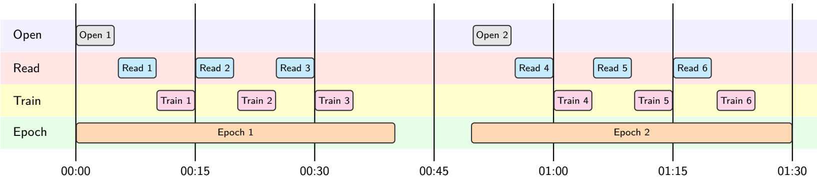


Figure 8.9: Naive data fetching implementation.

the system loads and prepares subsequent batches, maintaining a consistent supply of ready data (Martín Abadi et al. 2015).

Overlapping builds upon prefetching by coordinating multiple pipeline stages to execute concurrently. The system processes the current batch while simultaneously preparing future batches through data loading and preprocessing operations. This coordination establishes a continuous data flow through the training pipeline, as illustrated in Figure 8.10.

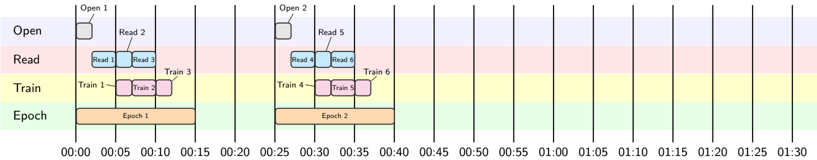


Figure 8.10: Parallel fetching and overlapping implementation. The job finishes at 00:40 seconds, instead of 01:30 seconds as in Figure 8.9.

These optimization techniques demonstrate particular value in scenarios involving large-scale datasets, preprocessing-intensive data, multi-GPU training configurations, or high-latency storage systems. The following section examines the specific mechanics of implementing these techniques in modern training systems.

8.5.1.1 Mechanics

Prefetching and overlapping optimize the training pipeline by enabling different stages of data processing and computation to operate concurrently rather than sequentially. These techniques maximize resource utilization by addressing bottlenecks in data transfer and preprocessing.

As you recall, training data undergoes three main stages: retrieval from storage, transformation into a suitable format, and utilization in model training. An unoptimized pipeline executes these stages sequentially. The GPU remains idle during data fetching and preprocessing, waiting for data preparation to complete. This sequential execution creates significant inefficiencies in the training process.

Prefetching eliminates waiting time by loading data asynchronously during model computation. Data loaders operate as separate threads or processes, preparing the next batch while the current batch trains. This ensures immediate data availability for the GPU when the current batch completes.

Overlapping extends this efficiency by coordinating all three pipeline stages simultaneously. As the GPU processes one batch, preprocessing begins on the next batch, while data fetching starts for the subsequent batch. This coordination maintains constant activity across all pipeline stages.

Modern machine learning frameworks implement these techniques through built-in utilities. PyTorch's `DataLoader` class demonstrates this implementation. An example of this usage is shown in Listing 8.2.

Listing 8.2: Using PyTorch DataLoader with batching and prefetching

```
loader = DataLoader(dataset,
                    batch_size=32,
                    num_workers=4,
                    prefetch_factor=2)
```

The parameters `num_workers` and `prefetch_factor` control parallel processing and data buffering. Multiple worker processes handle data loading and preprocessing concurrently, while prefetch_factor determines the number of batches prepared in advance.

Buffer management plays a key role in pipeline efficiency. The prefetch buffer size requires careful tuning to balance resource utilization. A buffer that is too small causes the GPU to wait for data preparation, reintroducing the idle time these techniques aim to eliminate. Conversely, allocating an overly large buffer consumes memory that could otherwise store model parameters or larger batch sizes.

The implementation relies on effective CPU-GPU coordination. The CPU manages data preparation tasks while the GPU handles computation. This division of labor, combined with storage I/O operations, creates an efficient pipeline that minimizes idle time across hardware resources.

These optimization techniques yield particular benefits in scenarios involving slow storage access, complex data preprocessing, or large datasets. The next section examines the specific advantages these techniques offer in different training contexts.

8.5.1.2 Benefits

Prefetching and overlapping are powerful techniques that significantly enhance the efficiency of training pipelines by addressing key bottlenecks in data handling and computation. To illustrate the impact of these benefits, Table 8.4 presents the following comparison:

Table 8.4: Comparison of training pipeline characteristics with and without prefetching and overlapping.

Aspect	Traditional Pipeline	With Prefetching & Overlapping
GPU Utilization	Frequent idle periods	Near-constant utilization
Training Time	Longer due to sequential operations	Reduced through parallelism
Resource Usage	Often suboptimal	Maximized across available hardware
Scalability	Limited by slowest component	Adaptable to various bottlenecks

One of the most critical advantages of these methods is the improvement in GPU utilization. In traditional, unoptimized pipelines, the GPU often remains idle while waiting for data to be fetched and preprocessed. This idle time creates inefficiencies, especially in workflows where data augmentation or preprocessing involves complex transformations. By introducing asynchronous data loading and overlapping, these techniques ensure that the GPU consistently has data ready to process, eliminating unnecessary delays.

Another important benefit is the reduction in overall training time. Prefetching and overlapping allow the computational pipeline to operate continuously, with multiple stages working simultaneously rather than sequentially. For example, while the GPU processes the current batch, the data loader fetches and preprocesses the next batch, ensuring a steady flow of data through the system. This parallelism minimizes latency between training iterations, allowing for faster completion of training cycles, particularly in scenarios involving large-scale datasets.

Additionally, these techniques are highly scalable and adaptable to various hardware configurations. Prefetching buffers and overlapping mechanisms can be tuned to match the specific requirements of a system, whether the bottleneck lies in slow storage, limited network bandwidth, or computational constraints. By aligning the data pipeline with the capabilities of the underlying hardware, prefetching and overlapping maximize resource utilization, making them invaluable for large-scale machine learning workflows.

Overall, prefetching and overlapping directly address some of the most common inefficiencies in training pipelines. By optimizing data flow and computation, these methods not only improve hardware efficiency but also enable the training of more complex models within shorter timeframes.

8.5.1.3 Use Cases

Prefetching and overlapping are highly versatile techniques that can be applied across various machine learning domains and tasks to enhance pipeline efficiency. Their benefits are most evident in scenarios where data handling and preprocessing are computationally expensive or where large-scale datasets create potential bottlenecks in data transfer and loading.

One of the primary use cases is in computer vision, where datasets often consist of high-resolution images requiring extensive preprocessing. Tasks such as image classification, object detection, or semantic segmentation typically involve operations like resizing, normalization, and data augmentation, all of which can significantly increase preprocessing time. By employing prefetching and overlapping, these operations can be carried out concurrently with computation, ensuring that the GPU remains busy during the training process.

For example, a typical image classification pipeline might include random cropping (10 ms), color jittering (15 ms), and normalization (5 ms). Without prefetching, these 30ms of preprocessing would delay each training step. Prefetching allows these operations to occur during the previous batch's computation.

Natural language processing (NLP) workflows also benefit from these techniques, particularly when working with large corpora of text data. For instance, preprocessing text data involves tokenization (converting words to numbers), padding sequences to equal length, and potentially subword tokenization. In a BERT model training pipeline, these steps might process thousands of sentences per batch. Prefetching allows this text processing to happen concurrently with model training. Prefetching ensures that these transformations occur in parallel with training, while overlapping optimizes data transfer and computation. This is especially useful in transformer-based models like BERT or GPT, which require consistent throughput to maintain efficiency given their high computational demand.

Distributed training systems, which we will discuss next, involve multiple GPUs or nodes, present another critical application for prefetching and overlapping. In distributed setups, network latency and data transfer rates often become the primary bottleneck. Prefetching mitigates these issues by ensuring that data is ready and available before it is required by any specific GPU. Overlapping further optimizes distributed training pipelines by coordinating the data preprocessing on individual nodes while the central computation continues, thus reducing overall synchronization delays.

Beyond these domains, prefetching and overlapping are particularly valuable in workflows involving large-scale datasets stored on remote or cloud-based systems. When training on cloud platforms, the data may need to be fetched over a network or from distributed storage, which introduces additional latency. Using prefetching and overlapping in such cases helps minimize the impact of these delays, ensuring that training proceeds smoothly despite slower data access speeds.

These use cases illustrate how prefetching and overlapping address inefficiencies in various machine learning pipelines. By optimizing the flow of data and computation, these techniques enable faster, more reliable training workflows across a wide range of applications.

8.5.1.4 Challenges and Trade-offs

While prefetching and overlapping are powerful techniques for optimizing training pipelines, their implementation comes with certain challenges and trade-offs. Understanding these limitations is crucial for effectively applying these methods in real-world machine learning workflows.

One of the primary challenges is the increased memory usage that accompanies prefetching and overlapping. By design, these techniques rely on maintaining a buffer of prefetched data batches, which requires additional memory resources. For large datasets or high-resolution inputs, this memory demand can become significant, especially when training on GPUs with limited memory capacity. If the buffer size is not carefully tuned, it may lead to out-of-memory errors, forcing practitioners to reduce batch sizes or adjust other parameters, which can impact overall efficiency.

For example, with a prefetch factor of 2 and batch size of 256 high-resolution images (1024×1024 pixels), the buffer might require an additional 2 GB of GPU memory. This becomes particularly challenging when training vision models that already require significant memory for their parameters and activations.

Another difficulty lies in tuning the parameters that control prefetching and overlapping. Settings such as `num_workers` and `prefetch_factor` in PyTorch, or buffer sizes in other frameworks, need to be optimized for the specific hardware and workload. For instance, increasing the number of worker threads can improve throughput up to a point, but beyond that, it may lead to contention for CPU resources or even degrade performance due to excessive context switching. Determining the optimal configuration often requires empirical testing, which can be time-consuming. A common starting point is to set `num_workers` to the number of CPU cores available. However, on a 16-core system processing large images, using all cores for data loading might leave insufficient CPU resources for other essential operations, potentially slowing down the entire pipeline.

Debugging also becomes more complex in pipelines that employ prefetching and overlapping. Asynchronous data loading and multithreading or multiprocessing introduce potential race conditions, deadlocks, or synchronization issues. Diagnosing errors in such systems can be challenging because the execution flow is no longer straightforward. Developers may need to invest additional effort into monitoring, logging, and debugging tools to ensure that the pipeline operates reliably.

Moreover, there are scenarios where prefetching and overlapping may offer minimal benefits. For instance, in systems where storage access or network bandwidth is significantly faster than the computation itself, these techniques might not noticeably improve throughput. In such cases, the additional complexity and memory overhead introduced by prefetching may not justify its use.

Finally, prefetching and overlapping require careful coordination across different components of the training pipeline, such as storage, CPUs, and GPUs. Poorly designed pipelines can lead to imbalances where one stage becomes a bottleneck, negating the advantages of these techniques. For example, if the data loading process is too slow to keep up with the GPU's processing speed, the benefits of overlapping will be limited.

Despite these challenges, prefetching and overlapping remain essential tools for optimizing training pipelines when used appropriately. By understanding and addressing their trade-offs, practitioners can implement these techniques effectively, ensuring smoother and more efficient machine learning workflows.

8.5.2 Mixed-Precision Training

Mixed-precision training combines different numerical precisions during model training to optimize computational efficiency. This approach uses combinations of 32-bit floating-point (FP32), 16-bit floating-point (FP16), and brain floating-point (bfloat16) formats to reduce memory usage and speed up computation while preserving model accuracy (Micikevicius et al. 2017a; Y. Wang and Kanwar 2019).

A neural network trained in FP32 requires 4 bytes per parameter, while both FP16 and bfloat16 use 2 bytes. For a model with 10^9 parameters, this reduction cuts memory usage from 4 GB to 2 GB. This memory reduction enables larger batch sizes and deeper architectures on the same hardware.

The numerical precision differences between these formats shape their use cases. FP32 represents numbers from approximately $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$ with 7 decimal digits of precision. FP16 ranges from $\pm 6.10 \times 10^{-5}$ to $\pm 65{,}504$ with 3-4 decimal digits of precision. Bfloat16, developed by Google Brain, maintains the same dynamic range as FP32 ($\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$)

but with reduced precision (3-4 decimal digits). This range preservation makes bfloat16 particularly suited for deep learning training, as it handles large and small gradients more effectively than FP16.

The hybrid approach proceeds in three main phases, as illustrated in Figure 8.11. During the forward pass, input data converts to reduced precision (FP16 or bfloat16), and matrix multiplications execute in this format, including activation function computations. In the gradient computation phase, the backward pass calculates gradients in reduced precision, but results are stored in FP32 master weights. Finally, during weight updates, the optimizer updates the main weights in FP32, and these updated weights convert back to reduced precision for the next forward pass.

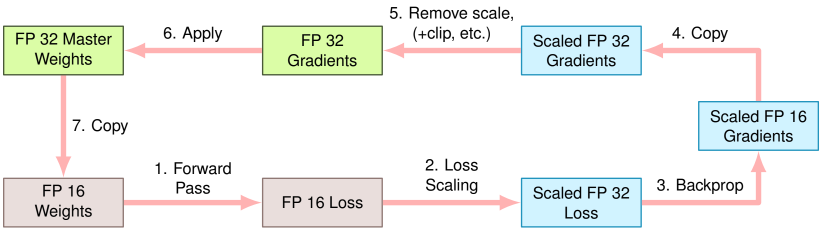


Figure 8.11: Mixed precision training flow.

Modern hardware architectures are specifically designed to accelerate reduced precision computations. GPUs from NVIDIA include Tensor Cores[9] optimized for FP16 and bfloat16 operations (Xianyan Jia et al. 2018). Google's TPUs natively support bfloat16, as this format was specifically designed for machine learning workloads. These architectural optimizations typically enable an order of magnitude higher computational throughput for reduced precision operations compared to FP32, making mixed-precision training particularly efficient on modern hardware.

8.5.2.1 FP16 Computation

The majority of operations in mixed-precision training, such as matrix multiplications and activation functions, are performed in FP16. The reduced precision allows these calculations to be executed faster and with less memory consumption compared to FP32. FP16 operations are particularly effective on modern GPUs equipped with Tensor Cores, which are designed to accelerate computations involving half-precision values. These cores perform FP16 operations natively, resulting in significant speedups.

8.5.2.2 FP32 Accumulation

While FP16 is efficient, its limited precision can lead to numerical instability, especially in critical operations like gradient updates. To mitigate this, mixed-precision training retains FP32 precision for certain steps, such as weight updates and gradient accumulation. By maintaining higher precision for these calculations, the system avoids the risk of gradient underflow or overflow, ensuring the model converges correctly during training.

[9] Tensor Cores: NVIDIA GPU units that accelerate matrix operations with reduced precision formats like FP16 and bfloat16, boosting deep learning performance by enabling parallel computations.

8.5.2.3 Loss Scaling

One of the key challenges with FP16 is its reduced dynamic range, which increases the likelihood of gradient values becoming too small to be represented accurately. Loss scaling addresses this issue by temporarily amplifying gradient values during backpropagation. Specifically, the loss value is scaled by a large factor (e.g., 2^{10}) before gradients are computed, ensuring they remain within the representable range of FP16. Once the gradients are computed, the scaling factor is reversed during the weight update step to restore the original gradient magnitude. This process allows FP16 to be used effectively without sacrificing numerical stability.

Modern machine learning frameworks, such as PyTorch and TensorFlow, provide built-in support for mixed-precision training. These frameworks abstract the complexities of managing different precisions, enabling practitioners to implement mixed-precision workflows with minimal effort. For instance, PyTorch's `torch.cuda.amp` (Automatic Mixed Precision) library automates the process of selecting which operations to perform in FP16 or FP32, as well as applying loss scaling when necessary.

Combining FP16 computation, FP32 accumulation, and loss scaling allows us to achieve mixed-precision training, resulting in a significant reduction in memory usage and computational overhead without compromising the accuracy or stability of the training process. The following sections will explore the practical advantages of this approach and its impact on modern machine learning workflows.

8.5.2.4 Benefits

Mixed-precision training offers several significant advantages that make it an essential optimization technique for modern machine learning workflows. By reducing memory usage and computational load, it enables practitioners to train larger models, process bigger batches, and achieve faster results, all while maintaining model accuracy and convergence.

One of the most prominent benefits of mixed-precision training is its substantial reduction in memory consumption. FP16 computations require only half the memory of FP32 computations, which directly reduces the storage required for activations, weights, and gradients during training. For instance, a transformer model with 1 billion parameters requires 4 GB of memory for weights in FP32, but only 2 GB in FP16. This memory efficiency allows for larger batch sizes, which can lead to more stable gradient estimates and faster convergence. Additionally, with less memory consumed per operation, practitioners can train deeper and more complex models on the same hardware, unlocking capabilities that were previously limited by memory constraints.[10]

Another key advantage is the acceleration of computations. Modern GPUs, such as those equipped with Tensor Cores, are specifically optimized for FP16 operations. These cores enable hardware to process more operations per cycle compared to FP32, resulting in faster training times. For matrix multiplication operations, which constitute 80-90% of training computation time in large models, FP16 can achieve 2-3× speedup compared to FP32. This computational speedup becomes particularly noticeable in large-scale models, such as

[10] Transformers are neural networks that use attention mechanisms to dynamically capture relationships between elements in sequential data. Unlike traditional architectures, transformers can process all sequence elements in parallel through multi-head attention, where each head learns different relationship patterns. This parallelization enables efficient processing of long sequences, making transformers particularly effective for tasks like language modeling and sequence translation.

transformers and convolutional neural networks, where matrix multiplications dominate the workload.

Mixed-precision training also improves hardware utilization by better matching the capabilities of modern accelerators. In traditional FP32 workflows, the computational throughput of GPUs is often underutilized due to their design for parallel processing. FP16 operations, being less demanding, allow more computations to be performed simultaneously, ensuring that the hardware operates closer to its full capacity.

Finally, mixed-precision training aligns well with the requirements of distributed and cloud-based systems. In distributed training, where large-scale models are trained across multiple GPUs or nodes, memory and bandwidth become critical constraints. By reducing the size of tensors exchanged between devices, mixed precision not only speeds up inter-device communication but also decreases overall resource demands. This makes it particularly effective in environments where scalability and cost-efficiency are priorities.

Overall, the benefits of mixed-precision training extend beyond performance improvements. By optimizing memory usage and computation, this technique empowers machine learning practitioners to train cutting-edge models more efficiently, making it a cornerstone of modern machine learning.

8.5.2.5 Use Cases

Mixed-precision training has become a essential in machine learning workflows, particularly in domains and scenarios where computational efficiency and memory optimization are critical. Its ability to enable faster training and larger model capacities makes it highly applicable across a variety of machine learning tasks and architectures.

One of the most prominent use cases is in training large-scale machine learning models. In natural language processing, models such as BERT (345M parameters), GPT-3 (175B parameters), and Transformer-based architectures involve extensive matrix multiplications and large parameter sets. Mixed-precision training allows these models to operate with larger batch sizes or deeper configurations, facilitating faster convergence and improved accuracy on massive datasets.

In computer vision, tasks such as image classification, object detection, and segmentation often require handling high-resolution images and applying computationally intensive convolutional operations. By leveraging mixed-precision training, these workloads can be executed more efficiently, enabling the training of advanced architectures like ResNet, EfficientNet, and vision transformers within practical resource limits.

Mixed-precision training is also particularly valuable in reinforcement learning (RL), where models interact with environments to optimize decision-making policies. RL often involves high-dimensional state spaces and requires substantial computational resources for both model training and simulation. Mixed precision reduces the overhead of these processes, allowing researchers to focus on larger environments and more complex policy networks.

Another critical application is in distributed training systems. When training models across multiple GPUs or nodes, memory and bandwidth become limiting factors for scalability. Mixed precision addresses these issues by reducing the size of activations, weights, and gradients exchanged between devices. For example, in a distributed training setup with 8 GPUs, reducing tensor sizes from FP32 to FP16 can halve the communication bandwidth requirements from 320 GB/s to 160 GB/s. This optimization is especially beneficial in cloud-based environments, where resource allocation and cost efficiency are paramount.

Additionally, mixed-precision training is increasingly used in areas such as speech processing, generative modeling, and scientific simulations. Models in these fields often have large data and parameter requirements that can push the limits of traditional FP32 workflows. By optimizing memory usage and leveraging the speedups provided by Tensor Cores, practitioners can train state-of-the-art models faster and more cost-effectively.

The adaptability of mixed-precision training to diverse tasks and domains underscores its importance in modern machine learning. Whether applied to large-scale natural language models, computationally intensive vision architectures, or distributed training environments, this technique empowers researchers and engineers to push the boundaries of what is computationally feasible.

8.5.2.6 Challenges and Trade-offs

While mixed-precision training offers significant advantages in terms of memory efficiency and computational speed, it also introduces several challenges and trade-offs that must be carefully managed to ensure successful implementation.

One of the primary challenges lies in the reduced precision of FP16. While FP16 computations are faster and require less memory, their limited dynamic range $(\pm 65,504)$ can lead to numerical instability, particularly during gradient computations. Small gradient values below 6×10^{-5} become too small to be represented accurately in FP16, resulting in underflow. While loss scaling addresses this by multiplying gradients by factors like 2^8 to 2^{14}, implementing and tuning this scaling factor adds complexity to the training process.

Another trade-off involves the increased risk of convergence issues. While many modern machine learning tasks perform well with mixed-precision training, certain models or datasets may require higher precision to achieve stable and reliable results. For example, recurrent neural networks with long sequences often accumulate numerical errors in FP16, requiring careful gradient clipping and precision management. In such cases, practitioners may need to experiment with selectively enabling or disabling FP16 computations for specific operations, which can complicate the training workflow.

Debugging and monitoring mixed-precision training also require additional attention. Numerical issues such as NaN (Not a Number) values in gradients or activations are more common in FP16 workflows and may be difficult to trace without proper tools and logging. For instance, gradient explosions in deep networks might manifest differently in mixed precision, appearing as infinities in FP16 before they would in FP32. Frameworks like PyTorch and TensorFlow provide utilities for debugging mixed-precision training, but these tools may not catch every edge case, especially in custom implementations.

Another challenge is the dependency on specialized hardware. Mixed-precision training relies heavily on GPU architectures optimized for FP16 operations, such as Tensor Cores in NVIDIA's GPUs. While these GPUs are becoming increasingly common, not all hardware supports mixed-precision operations, limiting the applicability of this technique in some environments.

Finally, there are scenarios where mixed-precision training may not provide significant benefits. Models with relatively low computational demand (less than 10M parameters) or small parameter sizes may not fully utilize the speedups offered by FP16 operations. In such cases, the additional complexity of mixed-precision workflows may outweigh their potential advantages.

Despite these challenges, mixed-precision training remains a highly effective optimization technique for most large-scale machine learning tasks. By understanding and addressing its trade-offs, practitioners can harness its benefits while minimizing potential drawbacks, ensuring efficient and reliable training workflows.

8.5.3 Gradient Accumulation and Checkpointing

Training large machine learning models often requires significant memory resources, particularly for storing three key components: activations (intermediate layer outputs), gradients (parameter updates), and model parameters (weights and biases) during forward and backward passes. However, memory constraints on GPUs can limit the batch size or the complexity of models that can be trained on a given device.

Gradient accumulation and activation checkpointing are two techniques designed to address these limitations by optimizing how memory is utilized during training. Both techniques enable researchers and practitioners to train larger and more complex models, making them indispensable tools for modern deep learning workflows. In the following sections, we will go deeper into the mechanics of gradient accumulation and activation checkpointing, exploring their benefits, use cases, and practical implementation.

8.5.3.1 Mechanics

Gradient accumulation and activation checkpointing operate on distinct principles, but both aim to optimize memory usage during training by modifying how forward and backward computations are handled.

Gradient Accumulation. Gradient accumulation simulates larger batch sizes by splitting a single effective batch into smaller "micro-batches." As illustrated in Figure 8.12, during each forward and backward pass, the gradients for a micro-batch are computed and added to an accumulated gradient buffer. Instead of immediately applying the gradients to update the model parameters, this process repeats for several micro-batches. Once the gradients from all micro-batches in the effective batch are accumulated, the parameters are updated using the combined gradients.

This process allows models to achieve the benefits of training with larger batch sizes, such as improved gradient estimates and convergence stability, without requiring the memory to store an entire batch at once. For instance, in PyTorch, this can be implemented by adjusting the learning rate proportionally to the number of accumulated micro-batches and calling `optimizer.step()` only after processing the entire effective batch.

The key steps in gradient accumulation are:

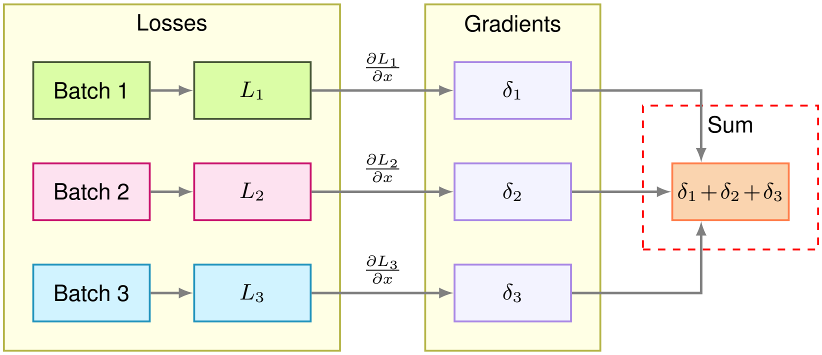

Losses
Gradients
Batch 1
L_1
$\frac{\partial L_1}{\partial x}$
δ_1
Sum
Batch 2
L_2
$\frac{\partial L_2}{\partial x}$
δ_2
$\delta_1 + \delta_2 + \delta_3$
Batch 3
L_3
$\frac{\partial L_3}{\partial x}$
δ_3

Figure 8.12: Gradient accumulation.

1. Perform the forward pass for a micro-batch.
2. Compute the gradients during the backward pass.
3. Accumulate the gradients into a buffer without updating the model parameters.
4. Repeat steps 1-3 for all micro-batches in the effective batch.
5. Update the model parameters using the accumulated gradients after all micro-batches are processed.

Activation Checkpointing. Activation checkpointing reduces memory usage during the backward pass by discarding and selectively recomputing activations. In standard training, activations from the forward pass are stored in memory for use in gradient computations during backpropagation. However, these activations can consume significant memory, particularly in deep networks.

With checkpointing, only a subset of the activations is retained during the forward pass. When gradients need to be computed during the backward pass, the discarded activations are recomputed on demand by re-executing parts of the forward pass, as illustrated in Figure 8.13. This approach trades computational efficiency for memory savings, as the recomputation increases training time but allows deeper models to be trained within limited memory constraints. The figure shows how memory is saved by avoiding storage of unnecessarily large intermediate tensors from the forward pass, and simply recomputing them on demand in the backwards pass.

The implementation involves:

1. Splitting the model into segments.
2. Retaining activations only at the boundaries of these segments during the forward pass.
3. Recomputing activations for intermediate layers during the backward pass when needed.

Frameworks like PyTorch provide tools such as `torch.utils.checkpoint` to simplify this process. Checkpointing is particularly effective for very deep architectures, such as transformers or large convolutional networks, where the memory required for storing activations can exceed the GPU’s capacity.

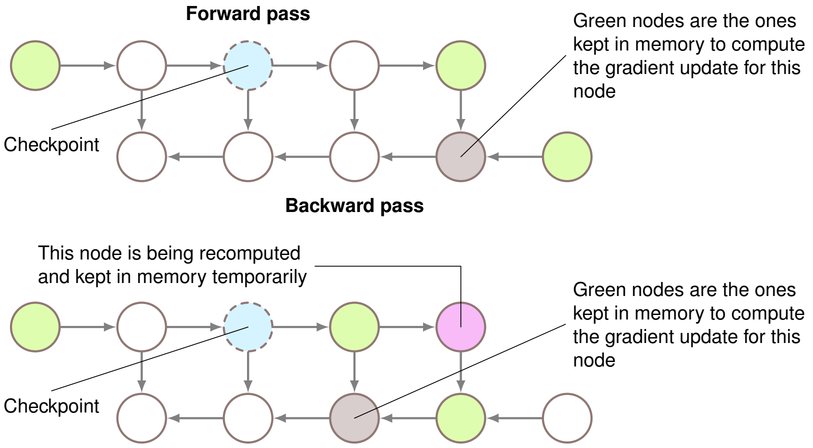

Forward pass
Green nodes are the ones kept in memory to compute the gradient update for this node
Checkpoint
Backward pass
This node is being recomputed and kept in memory temporarily
Green nodes are the ones kept in memory to compute the gradient update for this node
Checkpoint

Figure 8.13: Diagram showing how activation checkpointing helps to reduce memory usage during training.

The synergy between gradient accumulation and checkpointing enables training of larger, more complex models. Gradient accumulation manages memory constraints related to batch size, while checkpointing optimizes memory usage for intermediate activations. Together, these techniques expand the range of models that can be trained on available hardware.

8.5.3.2 Benefits

Gradient accumulation and activation checkpointing provide solutions to the memory limitations often encountered in training large-scale machine learning models. By optimizing how memory is used during training, these techniques enable the development and deployment of complex architectures, even on hardware with constrained resources.

One of the primary benefits of gradient accumulation is its ability to simulate larger batch sizes without increasing the memory requirements for storing the full batch. Larger batch sizes are known to improve gradient estimates, leading to more stable convergence and faster training. With gradient accumulation, practitioners can achieve these benefits while working with smaller micro-batches that fit within the GPU's memory. This flexibility is useful when training models on high-resolution data, such as large images or 3D volumetric data, where even a single batch may exceed available memory.

Activation checkpointing, on the other hand, significantly reduces the memory footprint of intermediate activations during the forward pass. This allows for the training of deeper models, which would otherwise be infeasible due to memory constraints. By discarding and recomputing activations as needed, checkpointing frees up memory that can be used for larger models, additional layers, or higher resolution data. This is especially important in state-of-the-art architectures, such as transformers or dense convolutional networks, which require substantial memory to store intermediate computations.

Both techniques enhance the scalability of machine learning workflows. In resource-constrained environments, such as cloud-based platforms or edge devices, these methods provide a means to train models efficiently without requiring expensive hardware upgrades. Furthermore, they enable researchers to experiment with larger and more complex architectures, pushing the boundaries of what is computationally feasible.

Beyond memory optimization, these techniques also contribute to cost efficiency. By reducing the hardware requirements for training, gradient accumulation and checkpointing lower the overall cost of development, making them valuable for organizations working within tight budgets. This is particularly relevant for startups, academic institutions, or projects running on shared computing resources.

Gradient accumulation and activation checkpointing provide both technical and practical advantages. These techniques create a more flexible, scalable, and cost-effective approach to training large-scale models, empowering practitioners to tackle increasingly complex machine learning challenges.

8.5.3.3 Use Cases

Gradient accumulation and activation checkpointing are particularly valuable in scenarios where hardware memory limitations present significant challenges during training. These techniques are widely used in training large-scale models, working with high-resolution data, and optimizing workflows in resource-constrained environments.

A common use case for gradient accumulation is in training models that require large batch sizes to achieve stable convergence. For example, models like GPT, BERT, and other transformer architectures often benefit from larger batch sizes due to their improved gradient estimates. However, these batch sizes can quickly exceed the memory capacity of GPUs, especially when working with high-dimensional inputs or multiple GPUs. By accumulating gradients over multiple smaller micro-batches, gradient accumulation enables the use of effective large batch sizes without exceeding memory limits. This is particularly beneficial for tasks like language modeling, sequence-to-sequence learning, and image classification, where batch size significantly impacts training dynamics.

Activation checkpointing enables training of deep neural networks with numerous layers or complex computations. In computer vision, architectures like ResNet-152, EfficientNet, and DenseNet require substantial memory to store intermediate activations during training. Checkpointing reduces this memory requirement through strategic recomputation of activations, making it possible to train these deeper architectures within GPU memory constraints.

In the domain of natural language processing, models like GPT-3 or T5, with hundreds of layers and billions of parameters, rely heavily on checkpointing to manage memory usage. These models often exceed the memory capacity of a single GPU, making checkpointing a necessity for efficient training. Similarly, in generative adversarial networks (GANs), which involve both generator and discriminator models, checkpointing helps manage the combined memory requirements of both networks during training.

Another critical application is in resource-constrained environments, such as edge devices or cloud-based platforms. In these scenarios, memory is often a limiting factor, and upgrading hardware may not always be a viable option. Gradient accumulation and checkpointing provide a cost-effective solution for training models on existing hardware, enabling efficient workflows without requiring additional investment in resources.

These techniques are also indispensable in research and experimentation. They allow practitioners to prototype and test larger and more complex models, exploring novel architectures that would otherwise be infeasible due to memory constraints. This is particularly valuable for academic researchers and startups operating within limited budgets.

Gradient accumulation and activation checkpointing solve fundamental challenges in training large-scale models within memory-constrained environments. These techniques have become essential tools for practitioners in natural language processing, computer vision, generative modeling, and edge computing, enabling broader adoption of advanced machine learning architectures.

8.5.3.4 Challenges and Trade-offs

While gradient accumulation and activation checkpointing are powerful tools for optimizing memory usage during training, their implementation introduces several challenges and trade-offs that must be carefully managed to ensure efficient and reliable workflows.

One of the primary trade-offs of activation checkpointing is the additional computational overhead it introduces. By design, checkpointing saves memory by discarding and recomputing intermediate activations during the backward pass. This recomputation increases the training time, as portions of the forward pass must be executed multiple times. For example, in a transformer model with 12 layers, if checkpoints are placed every 4 layers, each intermediate activation would need to be recomputed up to three times during the backward pass. The extent of this overhead depends on how the model is segmented for checkpointing and the computational cost of each segment. Practitioners must strike a balance between memory savings and the additional time spent on recomputation, which may affect overall training efficiency.

Gradient accumulation, while effective at simulating larger batch sizes, can lead to slower parameter updates. Since gradients are accumulated over multiple micro-batches, the model parameters are updated less frequently compared to training with full batches. This delay in updates can impact the speed of convergence, particularly in models sensitive to batch size dynamics. Additionally, gradient accumulation requires careful tuning of the learning rate. For instance, if accumulating gradients over 4 micro-batches to simulate a batch size of 128, the learning rate typically needs to be scaled up by a factor of 4 to maintain the same effective learning rate as training with full batches. The effective batch size increases with accumulation, necessitating proportional adjustments to the learning rate to maintain stable training.

Debugging and monitoring are also more complex when using these techniques. In activation checkpointing, errors may arise during recomputation, making it more difficult to trace issues back to their source. Similarly, gradient accumulation requires ensuring that gradients are correctly accumulated and reset after each effective batch, which can introduce bugs if not handled properly.

Another challenge is the increased complexity in implementation. While modern frameworks like PyTorch provide utilities to simplify gradient accumulation and checkpointing, effective use still requires understanding the underlying principles. For instance, activation checkpointing demands segmenting the model appropriately to minimize recomputation overhead while achieving meaningful memory savings. Improper segmentation can lead to suboptimal performance or excessive computational cost.

These techniques may also have limited benefits in certain scenarios. For example, if the computational cost of recomputation in activation checkpointing is too high relative to the memory savings, it may negate the advantages of the technique. Similarly, for models or datasets that do not require large batch sizes, the complexity introduced by gradient accumulation may not justify its use.

Despite these challenges, gradient accumulation and activation checkpointing remain indispensable for training large-scale models under memory constraints. By carefully managing their trade-offs and tailoring their application to specific workloads, practitioners can maximize the efficiency and effectiveness of these techniques.

8.5.4 Comparison

As summarized in Table 8.5, these techniques vary in their implementation complexity, hardware requirements, and impact on computation speed and memory usage. The selection of an appropriate optimization strategy depends on factors such as the specific use case, available hardware resources, and the nature of performance bottlenecks in the training process.

Table 8.5: High-level comparison of the three optimization strategies, highlighting their key aspects, benefits, and challenges.

Aspect	Prefetching and Overlapping	Mixed-Precision Training	Gradient Accumulation and Checkpointing
Primary Goal	Minimize data transfer delays and maximize GPU utilization	Reduce memory consumption and computational overhead	Overcome memory limitations during backpropagation and parameter updates
Key Mechanism	Asynchronous data loading and parallel processing	Combining FP16 and FP32 computations	Simulating larger batch sizes and selective activation storage
Memory Impact	Increases memory usage for prefetch buffer	Reduces memory usage by using FP16	Reduces memory usage for activations and gradients
Computation Speed	Improves by reducing idle time	Accelerates computations using FP16	May slow down due to recomputations in checkpointing
Scalability	Highly scalable, especially for large datasets	Enables training of larger models	Allows training deeper models on limited hardware
Hardware Requirements	Benefits from fast storage and multi-core CPUs	Requires GPUs with FP16 support (e.g., Tensor Cores)	Works on standard hardware
Implementation Complexity	Moderate (requires tuning of prefetch parameters)	Low to moderate (with framework support)	Moderate (requires careful segmentation and accumulation)
Main Benefits	Reduces training time, improves hardware utilization	Faster training, larger models, reduced memory usage	Enables larger batch sizes and deeper models
Primary Challenges	Tuning buffer sizes, increased memory usage	Potential numerical instability, loss scaling needed	Increased computational overhead, slower parameter updates
Ideal Use Cases	Large datasets, complex preprocessing	Large-scale models, especially in NLP and computer vision	Very deep networks, memory-constrained environments

While these three techniques represent core optimization strategies in machine learning, they are part of a larger optimization landscape. Other notable techniques include pipeline parallelism for multi-GPU training, dynamic batching for variable-length inputs, and quantization for inference optimization. Practitioners should evaluate their specific requirements, including model architecture, dataset characteristics, and hardware constraints, to select the most appropriate combination of optimization techniques for their use case.

8.6 Distributed Systems

Thus far, we have focused on ML training pipelines from a single-system perspective. However, training machine learning models often requires scaling beyond a single machine due to increasing model complexity and dataset sizes. The demand for computational power, memory, and storage can exceed the capacity of individual devices, especially in domains like natural language processing, computer vision, and scientific computing. Distributed training addresses this challenge by spreading the workload across multiple machines, which coordinate to train a single model efficiently.

Distributed training addresses this challenge by spreading the workload across multiple machines, which coordinate to train a single model efficiently. This process typically involves splitting the dataset into non-overlapping subsets, assigning each subset to a different GPU, and performing forward and backward passes independently on each device. Once gradients are computed on each GPU, they are synchronized and aggregated before updating the model parameters, ensuring that all devices learn in a consistent manner. Figure 8.14 illustrates this process, showing how input data is divided, assigned to multiple GPUs for computation, and later synchronized to update the model collectively.

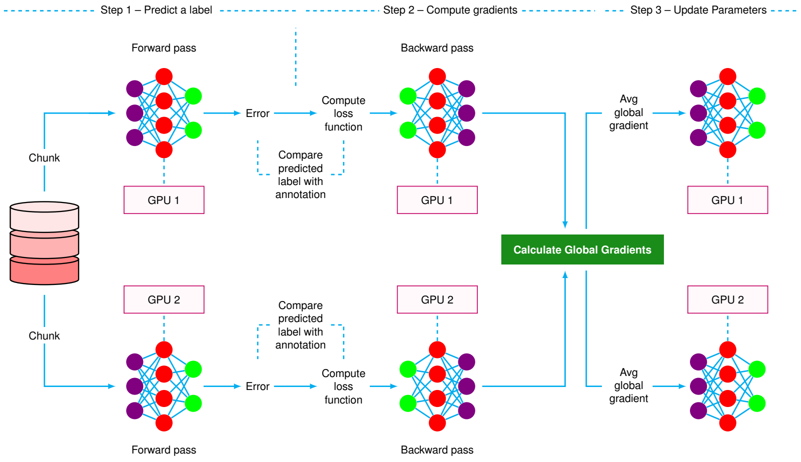


Figure 8.14: Multi-GPU training where the input data is divided into non-overlapping subsets, assigned to different GPUs for forward and backward passes, and then synchronized to aggregate gradients before updating model parameters.

This coordination introduces several fundamental challenges that distributed training systems must address. A distributed training system must orchestrate multi-machine computation by splitting up the work, managing communication between machines, and maintaining synchronization throughout the training process. Understanding these basic requirements provides the foundation for examining the main approaches to distributed training: data parallelism, which divides the training data across machines; model parallelism, which splits the model itself; and hybrid approaches that combine both strategies.

8.6.1 Data Parallelism

Data parallelism is a method for distributing the training process across multiple devices by splitting the dataset into smaller subsets. Each device trains a complete copy of the model using its assigned subset of the data. For example, when training an image classification model on 1 million images using 4 GPUs, each GPU would process 250,000 images while maintaining an identical copy of the model architecture.

It is particularly effective when the dataset size is large but the model size is manageable, as each device must store a full copy of the model in memory. This method is widely used in scenarios such as image classification and natural language processing, where the dataset can be processed in parallel without dependencies between data samples. For instance, when training a ResNet model on ImageNet, each GPU can independently process its portion of images since the classification of one image doesn't depend on the results of another.

Data parallelism builds on a key insight from stochastic gradient descent. Gradients computed on different minibatches can be averaged. This property enables parallel computation across devices. Let's examine why this works mathematically.

Consider a model with parameters θ training on a dataset D. The loss function for a single data point x_i is $L(\theta, x_i)$. In standard SGD with batch size B, the gradient update for a minibatch is:

$$g = \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta L(\theta, x_i)$$

In data parallelism with N devices, each device k computes gradients on its own minibatch B_k:

$$g_k = \frac{1}{|B_k|} \sum_{x_i \in B_k} \nabla_\theta L(\theta, x_i)$$

The global update averages these local gradients:

$$g_{\text{global}} = \frac{1}{N} \sum_{k=1}^{N} g_k$$

This averaging is mathematically equivalent to computing the gradient on the combined batch $B_{\text{total}} = \bigcup_{k=1}^{N} B_k$:

$$g_{\text{global}} = \frac{1}{|B_{\text{total}}|} \sum_{x_i \in B_{\text{total}}} \nabla_\theta L(\theta, x_i)$$

This equivalence shows why data parallelism maintains the statistical properties of SGD training. The approach distributes distinct data subsets across devices, computes local gradients independently, and averages these gradients to approximate the full-batch gradient.

The method parallels gradient accumulation, where a single device accumulates gradients over multiple forward passes before updating parameters. Both techniques leverage the additive properties of gradients to process large batches efficiently.

8.6.1.1 Mechanics

The process of data parallelism can be broken into a series of distinct steps, each with its role in ensuring the system operates efficiently. These steps are illustrated in Figure 8.15.

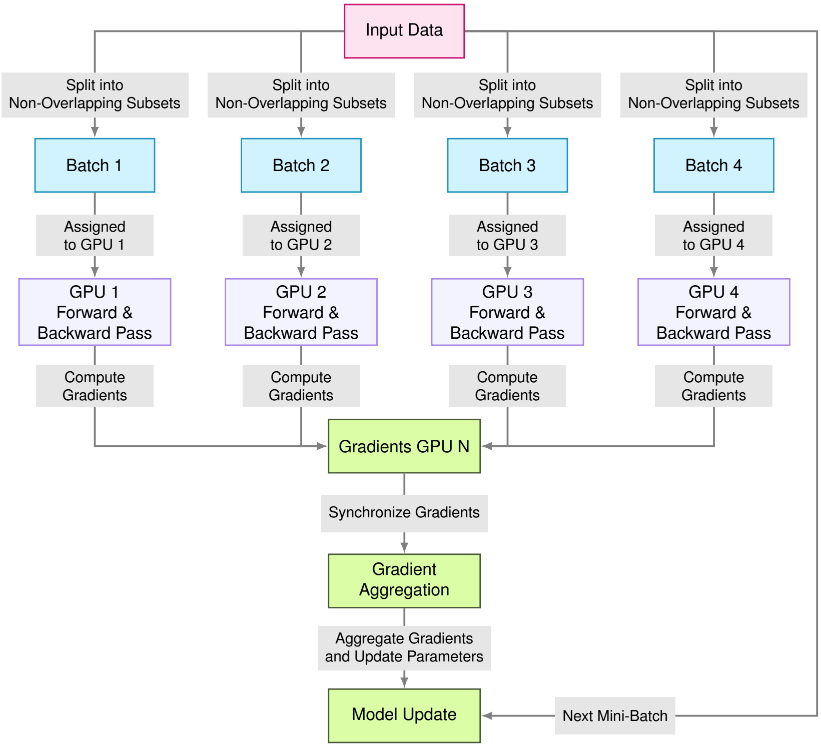

Input Data
Split into Non-Overlapping Subsets
Split into Non-Overlapping Subsets
Split into Non-Overlapping Subsets
Split into Non-Overlapping Subsets
Batch 1
Batch 2
Batch 3
Batch 4
Assigned to GPU 1
Assigned to GPU 2
Assigned to GPU 3
Assigned to GPU 4
GPU 1 Forward & Backward Pass
GPU 2 Forward & Backward Pass
GPU 3 Forward & Backward Pass
GPU 4 Forward & Backward Pass
Compute Gradients
Compute Gradients
Compute Gradients
Compute Gradients
Gradients GPU N
Synchronize Gradients
Gradient Aggregation
Aggregate Gradients and Update Parameters
Model Update
Next Mini-Batch

Figure 8.15: Data-level parallelism.

Dataset Splitting. The first step in data parallelism involves dividing the dataset into smaller, non-overlapping subsets. This ensures that each device processes a unique portion of the data, avoiding redundancy and enabling efficient utilization of available hardware. For instance, with a dataset of 100,000 training examples and 4 GPUs, each GPU would be assigned 25,000 examples. Modern frameworks like PyTorch's DistributedSampler handle this distribution automatically, implementing prefetching and caching mechanisms to ensure data is readily available for processing.

Device Forward Pass. Once the data subsets are distributed, each device performs the forward pass independently. During this stage, the model processes its assigned batch of data, generating predictions and calculating the loss. For example, in a ResNet-50 model, each GPU would independently compute the convolutions, activations, and final loss for its batch. The forward pass is computationally intensive and benefits from hardware accelerators like NVIDIA V100 GPUs or Google TPUs, which are optimized for matrix operations.

Backward Pass and Calculation. Following the forward pass, each device computes the gradients of the loss with respect to the model's parameters during the backward pass. Modern frameworks like PyTorch and TensorFlow handle this automatically through their autograd systems. For instance, if a model has 50 million parameters, each device calculates gradients for all parameters but based only on its local data subset.

Gradient Synchronization. To maintain consistency across the distributed system, the gradients computed by each device must be synchronized. This step typically uses the ring all-reduce algorithm, where each GPU communicates only with its neighbors, reducing communication overhead. For example, with 8 GPUs, each sharing gradients for a 100MB model, ring all-reduce requires only 7 communication steps instead of the 56 steps needed for naive peer-to-peer synchronization.

Parameter Updating. After gradient aggregation[11], each device independently updates model parameters using the chosen optimization algorithm, such as SGD with momentum or Adam. This decentralized update strategy, implemented in frameworks like PyTorch's DistributedDataParallel (DDP), enables efficient parameter updates without requiring a central coordination server. Since all devices have identical gradient values after synchronization, they perform mathematically equivalent updates to maintain model consistency across the distributed system.

For example, in a system with 8 GPUs training a ResNet model, each GPU computes local gradients based on its data subset. After gradient averaging via ring all-reduce, every GPU has the same global gradient values. Each device then independently applies these gradients using the optimizer's update rule. If using SGD with learning rate 0.1, the update would be `weights = weights - 0.1 * gradients`. This process maintains mathematical equivalence to single-device training while enabling distributed computation.

This process, which involves splitting data, performing computations, synchronizing results, and updating parameters, repeats for each batch of data. Modern frameworks automate this cycle, allowing developers to focus on model architecture and hyperparameter tuning rather than distributed computing logistics.

8.6.1.2 Benefits

Data parallelism offers several key benefits that make it the predominant approach for distributed training. By splitting the dataset across multiple devices and allowing each device to train an identical copy of the model, this approach effectively addresses the core challenges in modern AI training systems.

[11] The choice between summing or averaging gradients impacts model training dynamics. Gradient summation requires scaling the learning rate by the number of workers to maintain consistent update magnitudes. While gradient averaging provides more stable updates with reduced variance, it requires a central coordination node that can become a bottleneck as the number of workers increases. The decision depends on the specific distributed training setup and optimization goals.

The primary advantage of data parallelism is its linear scaling capability with large datasets. As datasets grow into the terabyte range, processing them on a single machine becomes prohibitively time-consuming. For example, training a vision transformer on ImageNet (1.2 million images) might take weeks on a single GPU, but only days when distributed across 8 GPUs. This scalability is particularly valuable in domains like language modeling, where datasets can exceed billions of tokens.

Hardware utilization efficiency represents another crucial benefit. Data parallelism maintains high GPU utilization rates, typically, above 85%, by ensuring each device actively processes its data portion. Modern implementations achieve this through asynchronous data loading and gradient computation overlapping with communication. For instance, while one batch computes gradients, the next batch's data is already being loaded and preprocessed.

Implementation simplicity sets data parallelism apart from other distribution strategies. Modern frameworks have reduced complex distributed training to just a few lines of code. For example, converting a PyTorch model to use data parallelism often requires only wrapping it in `DistributedDataParallel` and initializing a distributed environment. This accessibility has contributed significantly to its widespread adoption in both research and industry.

The approach also offers remarkable flexibility across model architectures. Whether training a ResNet (vision), BERT (language), or Graph Neural Network (graph data), the same data parallelism principles apply without modification. This universality makes it particularly valuable as a default choice for distributed training.

Training time reduction is perhaps the most immediate benefit. Given proper implementation, data parallelism can achieve near-linear speedup with additional devices. Training that takes 100 hours on a single GPU might complete in roughly 13 hours on 8 GPUs, assuming efficient gradient synchronization and minimal communication overhead.

While these benefits make data parallelism compelling, it's important to note that achieving these advantages requires careful system design. The next section examines the challenges that must be addressed to fully realize these benefits.

8.6.1.3 Challenges

While data parallelism is a powerful approach for distributed training, it introduces several challenges that must be addressed to achieve efficient and scalable training systems. These challenges stem from the inherent trade-offs between computation and communication, as well as the limitations imposed by hardware and network infrastructures.

Communication overhead represents the most significant bottleneck in data parallelism. During gradient synchronization, each device must exchange gradient updates—often hundreds of megabytes per step for large models. With 8 GPUs training a 1-billion-parameter model, each synchronization step might require transferring several gigabytes of data across the network. While high-speed interconnects like NVLink (300 GB/s) or InfiniBand (200 Gb/s) help, the overhead remains substantial. NCCL's ring-allreduce[12] algorithm reduces

[12] A communication strategy that minimizes data transfer overhead by organizing devices in a ring topology, first introduced for distributed machine learning in Horovod.

this burden by organizing devices in a ring topology, but communication costs still grow with model size and device count.

Scalability limitations become apparent as device count increases. While 8 GPUs might achieve 7× speedup (87.5% scaling efficiency), scaling to 64 GPUs typically yields only 45-50× speedup (70-78% efficiency) due to growing synchronization costs. This non-linear scaling means that doubling the number of devices rarely halves the training time, particularly in configurations exceeding 16-32 devices.

Memory constraints present a hard limit for data parallelism. Consider a transformer model with 175 billion parameters—it requires approximately 350 GB just to store model parameters in FP32. When accounting for optimizer states and activation memories, the total requirement often exceeds 1 TB per device. Since even high-end GPUs typically offer 80 GB or less, such models cannot use pure data parallelism.

Workload imbalance affects heterogeneous systems significantly. In a cluster mixing A100 and V100 GPUs, the A100s might process batches 1.7× faster, forcing them to wait for the V100s to catch up. This idle time can reduce cluster utilization by 20-30% without proper load balancing mechanisms.

Finally, there are challenges related to implementation complexity in distributed systems. While modern frameworks abstract much of the complexity, implementing data parallelism at scale still requires significant engineering effort. Ensuring fault tolerance, debugging synchronization issues, and optimizing data pipelines are non-trivial tasks that demand expertise in both machine learning and distributed systems.

Despite these challenges, data parallelism remains an important technique for distributed training, with many strategies available to address its limitations. In the next section, we will explore model parallelism, another strategy for scaling training that is particularly well-suited for handling extremely large models that cannot fit on a single device.

8.6.2 Model Parallelism

Model parallelism splits neural networks across multiple computing devices when the model's parameters exceed single-device memory limits. Unlike data parallelism, where each device contains a complete model copy, model parallelism assigns different model components to different devices (Shazeer, Mirhoseini, Maziarz, Davis, et al. 2017).

Several implementations of model parallelism exist. In layer-based splitting, devices process distinct groups of layers sequentially. For instance, the first device might compute layers 1-4 while the second handles layers 5-8. Channel-based splitting divides the channels within each layer across devices, such as the first device processing 512 channels while the second manages the remaining ones. For transformer architectures, attention head splitting distributes different attention heads to separate devices.

This distribution method enables training of large-scale models. GPT-3, with 175 billion parameters, relies on model parallelism for training. Vision transformers processing high-resolution 16k × 16k pixel images use model parallelism to manage memory constraints. Mixture-of-Expert architectures leverage this approach to distribute their conditional computation paths across hardware.

Device coordination follows a specific pattern during training. In the forward pass, data flows sequentially through model segments on different devices. The backward pass propagates gradients in reverse order through these segments. During parameter updates, each device modifies only its assigned portion of the model. This coordination ensures mathematical equivalence to training on a single device while enabling the handling of models that exceed individual device memory capacities.

8.6.2.1 Mechanics

Model parallelism divides neural networks across multiple computing devices, with each device computing a distinct portion of the model's operations. This division allows training of models whose parameter counts exceed single-device memory capacity. The technique encompasses device coordination, data flow management, and gradient computation across distributed model segments. The mechanics of model parallelism are illustrated in Figure 8.16. These steps are described next:

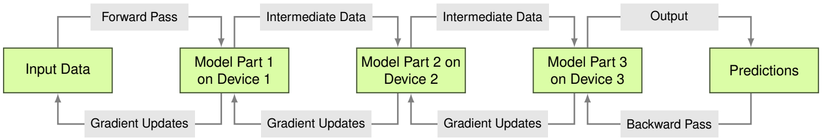


Figure 8.16: Model-level parallelism.

Model Partitioning. The first step in model parallelism is dividing the model into smaller segments. For instance, in a deep neural network, layers are often divided among devices. In a system with two GPUs, the first half of the layers might reside on GPU 1, while the second half resides on GPU 2. Another approach is to split computations within a single layer, such as dividing matrix multiplications in transformer models across devices.

Model Forward Pass. During the forward pass, data flows sequentially through the partitions. For example, data processed by the first set of layers on GPU 1 is sent to GPU 2 for processing by the next set of layers. This sequential flow ensures that the entire model is used, even though it is distributed across multiple devices. Efficient inter-device communication is crucial to minimize delays during this step (Research 2021).

Backward Pass and Calculation. The backward pass computes gradients through the distributed model segments in reverse order. Each device calculates local gradients for its parameters and propagates necessary gradient information to previous devices. In transformer models, this means backpropagating through attention computations and feed-forward networks across device boundaries.

For example, in a two-device setup with attention mechanisms split between devices, the backward computation works as follows: The second device computes gradients for the final feed-forward layers and attention heads. It then sends the gradient tensors for the attention output to the first device. The first device uses these received gradients to compute updates for its attention parameters and earlier layer weights.

Parameter Updates. Parameter updates occur independently on each device using the computed gradients and an optimization algorithm. A device holding attention layer parameters applies updates using only the gradients computed for those specific parameters. This localized update approach differs from data parallelism, which requires gradient averaging across devices.

The optimization step proceeds as follows: Each device applies its chosen optimizer (such as Adam or AdaFactor) to update its portion of the model parameters. A device holding the first six transformer layers updates only those layers' weights and biases. This local parameter update eliminates the need for cross-device synchronization during the optimization step, reducing communication overhead.

Iterative Process. Like other training strategies, model parallelism repeats these steps for every batch of data. As the dataset is processed over multiple iterations, the distributed model converges toward optimal performance.

Parallelism Variations. Model parallelism can be implemented through different strategies for dividing the model across devices. The three primary approaches are layer-wise partitioning, operator-level partitioning, and pipeline parallelism, each suited to different model structures and computational needs.

Layer-wise Partitioning. Layer-wise partitioning assigns distinct model layers to separate computing devices. In transformer architectures, this translates to specific devices managing defined sets of attention and feed-forward blocks. As illustrated in Figure 8.17, a 24-layer transformer model distributed across four devices assigns six consecutive transformer blocks to each device. Device 1 processes blocks 1-6, device 2 handles blocks 7-12, and so forth.

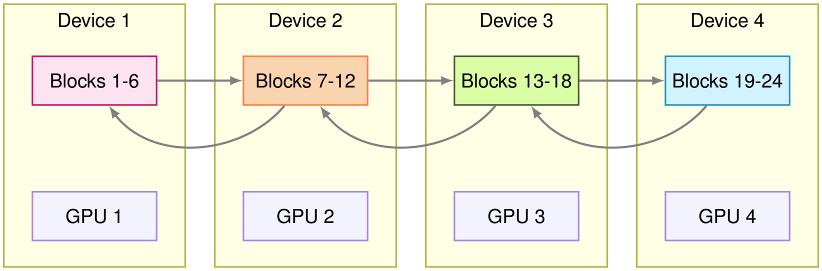
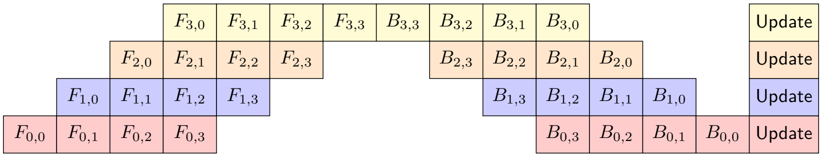


Figure 8.17: Example of pipeline parallelism.

This sequential processing introduces device idle time, as each device must wait for the previous device to complete its computation before beginning work. For example, while device 1 processes the initial blocks, devices 2, 3, and 4 remain inactive. Similarly, when device 2 begins its computation, device 1 sits idle. This pattern of waiting and idle time reduces hardware utilization efficiency compared to other parallelization strategies.

Layer-wise partitioning assigns distinct model layers to separate computing devices. In transformer architectures, this translates to specific devices managing defined sets of attention and feed-forward blocks. A 24-layer transformer model distributed across four devices assigns six consecutive transformer blocks to each device. Device 1 processes blocks 1-6, device 2 handles blocks 7-12, and so forth.

Pipeline Parallelism. Pipeline parallelism extends layer-wise partitioning by introducing microbatching to minimize device idle time, as illustrated in Figure 8.18. Instead of waiting for an entire batch to sequentially pass through all devices, the computation is divided into smaller segments called microbatches [harlap2018pipedream]. Each device, as represented by the rows in the drawing, processes its assigned model layers for different microbatches simultaneously. For example, the forward pass involves devices passing activations to the next stage (e.g., $F_{0,0}$ to $F_{1,0}$). The backward pass transfers gradients back through the pipeline (e.g., $B_{3,3}$ to $B_{2,3}$). This overlapping computation reduces idle time and increases throughput while maintaining the logical sequence of operations across devices.



			$F_{3,0}$	$F_{3,1}$	$F_{3,2}$	$F_{3,3}$	$B_{3,3}$	$B_{3,2}$	$B_{3,1}$	$B_{3,0}$				Update
		$F_{2,0}$	$F_{2,1}$	$F_{2,2}$	$F_{2,3}$			$B_{2,3}$	$B_{2,2}$	$B_{2,1}$	$B_{2,0}$			Update
	$F_{1,0}$	$F_{1,1}$	$F_{1,2}$	$F_{1,3}$					$B_{1,3}$	$B_{1,2}$	$B_{1,1}$	$B_{1,0}$		Update
$F_{0,0}$	$F_{0,1}$	$F_{0,2}$	$F_{0,3}$							$B_{0,3}$	$B_{0,2}$	$B_{0,1}$	$B_{0,0}$	Update

Figure 8.18: Example of pipeline parallelism.

In a transformer model distributed across four devices, device 1 would process blocks 1-6 for microbatch $N+1$ while device 2 computes blocks 7-12 for microbatch N. Simultaneously, device 3 executes blocks 13-18 for microbatch $N-1$, and device 4 processes blocks 19-24 for microbatch $N-2$. Each device maintains its assigned transformer blocks but operates on a different microbatch, creating a continuous flow of computation.

The transfer of hidden states between devices occurs continuously rather than in distinct phases. When device 1 completes processing a microbatch, it immediately transfers the output tensor of shape [microbatch_size, sequence_length, hidden_dimension] to device 2 and begins processing the next microbatch. This overlapping computation pattern maintains full hardware utilization while preserving the model's mathematical properties.

Operator-level Parallelism. Operator-level parallelism divides individual neural network operations across devices. In transformer models, this often means splitting attention computations. Consider a transformer with 64 attention heads and a hidden dimension of 4096. Two devices might split this computation as follows: Device 1 processes attention heads 1-32, computing queries, keys, and values for its assigned heads. Device 2 simultaneously processes heads 33-64. Each device handles attention computations for [batch_size, sequence_length, 2048] dimensional tensors.

Matrix multiplication operations in feed-forward networks also benefit from operator-level splitting. A feed-forward layer with input dimension 4096 and intermediate dimension 16384 can split across devices along the intermediate dimension. Device 1 computes the first 8192 intermediate features, while device

2 computes the remaining 8192 features. This division reduces peak memory usage while maintaining mathematical equivalence to the original computation.

Summary. Each of these partitioning methods addresses specific challenges in training large models, and their applicability depends on the model architecture and the resources available. By selecting the appropriate strategy, practitioners can train models that exceed the limits of individual devices, enabling the development of cutting-edge machine learning systems.

8.6.2.2 Benefits

Model parallelism offers several significant benefits, making it an essential strategy for training large-scale models that exceed the capacity of individual devices. These advantages stem from its ability to partition the workload across multiple devices, enabling the training of more complex and resource-intensive architectures.

Memory scaling represents the primary advantage of model parallelism. Current transformer architectures contain up to hundreds of billions of parameters. A 175 billion parameter model with 32-bit floating point precision requires 700 GB of memory just to store its parameters. When accounting for activations, optimizer states, and gradients during training, the memory requirement multiplies several fold. Model parallelism makes training such architectures feasible by distributing these memory requirements across devices.

Another key advantage is the efficient utilization of device memory and compute power. Since each device only needs to store and process a portion of the model, memory usage is distributed across the system. This allows practitioners to work with larger batch sizes or more complex layers without exceeding memory limits, which can also improve training stability and convergence.

Model parallelism also provides flexibility for different model architectures. Whether the model is sequential, as in many natural language processing tasks, or composed of computationally intensive operations, as in attention-based models or convolutional networks, there is a partitioning strategy that fits the architecture. This adaptability makes model parallelism applicable to a wide variety of tasks and domains.

Finally, model parallelism is a natural complement to other distributed training strategies, such as data parallelism and pipeline parallelism. By combining these approaches, it becomes possible to train models that are both large in size and require extensive data. This hybrid flexibility is especially valuable in cutting-edge research and production environments, where scaling models and datasets simultaneously is critical for achieving state-of-the-art performance.

While model parallelism introduces these benefits, its effectiveness depends on the careful design and implementation of the partitioning strategy. In the next section, we will discuss the challenges associated with model parallelism and the trade-offs involved in its use.

8.6.2.3 Challenges

While model parallelism provides a powerful approach for training large-scale models, it also introduces unique challenges. These challenges arise from the complexity of partitioning the model and the dependencies between partitions during training. Addressing these issues requires careful system design and optimization.

One major challenge in model parallelism is balancing the workload across devices. Not all parts of a model require the same amount of computation. For instance, in layer-wise partitioning, some layers may perform significantly more operations than others, leading to an uneven distribution of work. Devices responsible for the heavier computations may become bottlenecks, leaving others underutilized. This imbalance reduces overall efficiency and slows down training. Identifying optimal partitioning strategies is critical to ensuring all devices contribute evenly.

Another challenge is data dependency between devices. During the forward pass, activation tensors of shape [batch_size, sequence_length, hidden_dimension] must transfer between devices. For a typical transformer model with batch size 32, sequence length 2048, and hidden dimension 2048, each transfer moves approximately 512 MB of data at float32 precision. With gradient transfers in the backward pass, a single training step can require several gigabytes of inter-device communication. On systems using PCIe interconnects with 64 GB/s theoretical bandwidth, these transfers introduce significant latency.

Model parallelism also increases the complexity of implementation and debugging. Partitioning the model, ensuring proper data flow, and synchronizing gradients across devices require detailed coordination. Errors in any of these steps can lead to incorrect gradient updates or even model divergence. Debugging such errors is often more difficult in distributed systems, as issues may arise only under specific conditions or workloads.

A further challenge is pipeline bubbles in pipeline parallelism. With m pipeline stages, the first $m-1$ steps operate at reduced efficiency as the pipeline fills. For example, in an 8-device pipeline, the first device begins processing immediately, but the eighth device remains idle for 7 steps. This warmup period reduces hardware utilization by approximately $(m-1)/b$ percent, where b is the number of batches in the training step.

Finally, model parallelism may be less effective for certain architectures, such as models with highly interdependent operations. In these cases, splitting the model may lead to excessive communication overhead, outweighing the benefits of parallel computation. For such models, alternative strategies like data parallelism or hybrid approaches might be more suitable.

Despite these challenges, model parallelism remains an indispensable tool for training large models. With careful optimization and the use of modern frameworks, many of these issues can be mitigated, enabling efficient distributed training at scale.

8.6.3 Hybrid Parallelism

Hybrid parallelism combines model parallelism and data parallelism when training neural networks (D. Narayanan et al. 2021b). A model might be too large to store on one GPU (requiring model parallelism) while simultaneously needing to process large batches of data efficiently (requiring data parallelism).

Training a 175-billion parameter language model on a dataset of 300 billion tokens demonstrates hybrid parallelism in practice. The neural network layers distribute across multiple GPUs through model parallelism, while data parallelism enables different GPU groups to process separate batches. The hybrid approach coordinates these two forms of parallelization.

This strategy addresses two fundamental constraints. First, memory constraints arise when model parameters exceed single-device memory capacity. Second, computational demands increase when dataset size necessitates distributed processing.

8.6.3.1 Mechanics

Hybrid parallelism operates by combining the processes of model partitioning and dataset splitting, ensuring efficient utilization of both memory and computation across devices. This integration allows large-scale machine learning systems to overcome the constraints imposed by individual parallelism strategies.

Model and Data Partitioning. Hybrid parallelism divides both model architecture and training data across devices. The model divides through layer-wise or operator-level partitioning, where GPUs process distinct neural network segments. Simultaneously, the dataset splits into subsets, allowing each device group to train on different batches. A transformer model might distribute its attention layers across four GPUs, while each GPU group processes a unique 1,000-example batch. This dual partitioning distributes memory requirements and computational workload.

Forward Pass. During the forward pass, input data flows through the distributed model. Each device processes its assigned portion of the model using the data subset it holds. For example, in a hybrid system with four devices, two devices might handle different layers of the model (model parallelism) while simultaneously processing distinct data batches (data parallelism). Communication between devices ensures that intermediate outputs from model partitions are passed seamlessly to subsequent partitions.

Backward Pass and Gradient Calculation. During the backward pass, gradients are calculated for the model partitions stored on each device. Data-parallel devices that process the same subset of the model but different data batches aggregate their gradients, ensuring that updates reflect contributions from the entire dataset. For model-parallel devices, gradients are computed locally and passed to the next layer in reverse order. In a two-device model-parallel configuration, for example, the first device computes gradients for layers 1-3, then transmits these to the second device for layers 4-6. This combination of gradient synchronization and inter-device communication ensures consistency across the distributed system.

Parameter Updates. After gradient synchronization, model parameters are updated using the chosen optimization algorithm. Devices working in data parallelism update their shared model partitions consistently, while model-parallel devices apply updates to their local segments. Efficient communication is critical in this step to minimize delays and ensure that updates are correctly propagated across all devices.

Iterative Process. Hybrid parallelism follows an iterative process similar to other training strategies. The combination of model and data distribution allows the system to process large datasets and complex models efficiently over multiple training epochs. By balancing the computational workload and memory requirements, hybrid parallelism enables the training of advanced machine learning models that would otherwise be infeasible.

Parallelism Variations. Hybrid parallelism can be implemented in different configurations, depending on the model architecture, dataset characteristics, and available hardware. These variations allow for tailored solutions that optimize performance and scalability for specific training requirements.

Hierarchical Parallelism. Hierarchical hybrid parallelism applies model parallelism to divide the model across devices first and then layers data parallelism on top to handle the dataset distribution. For example, in a system with eight devices, four devices may hold different partitions of the model, while each partition is replicated across the other four devices for data parallel processing. This approach is well-suited for large models with billions of parameters, where memory constraints are a primary concern.

Hierarchical hybrid parallelism ensures that the model size is distributed across devices, reducing memory requirements, while data parallelism ensures that multiple data samples are processed simultaneously, improving throughput. This dual-layered approach is particularly effective for models like transformers, where each layer may have a significant memory footprint.

Intra-layer Parallelism. Intra-layer hybrid parallelism combines model and data parallelism within individual layers of the model. For instance, in a transformer architecture, the attention mechanism can be split across multiple devices (model parallelism), while each device processes distinct batches of data (data parallelism). This fine-grained integration allows the system to optimize resource usage at the level of individual operations, enabling training for models with extremely large intermediate computations.

This variation is particularly useful in scenarios where specific layers, such as attention or feedforward layers, have computationally intensive operations that are difficult to distribute effectively using model or data parallelism alone. Intra-layer hybrid parallelism addresses this challenge by applying both strategies simultaneously.

Inter-layer Parallelism. Inter-layer hybrid parallelism focuses on distributing the workload between model and data parallelism at the level of distinct model layers. For example, early layers of a neural network may be distributed using model parallelism, while later layers leverage data parallelism. This approach aligns with the observation that certain layers in a model may be more memory-intensive, while others benefit from increased data throughput.

This configuration allows for dynamic allocation of resources, adapting to the specific demands of different layers in the model. By tailoring the parallelism strategy to the unique characteristics of each layer, inter-layer hybrid parallelism achieves an optimal balance between memory usage and computational efficiency.

8.6.3.2 Benefits

The adoption of hybrid parallelism in machine learning systems addresses some of the most significant challenges posed by the ever-growing scale of models and datasets. By blending the strengths of model parallelism and data parallelism, this approach provides a comprehensive solution to scaling modern machine learning workloads.

One of the most prominent benefits of hybrid parallelism is its ability to scale seamlessly across both the model and the dataset. Modern neural networks, particularly transformers used in natural language processing and vision applications, often contain billions of parameters. These models, paired with massive datasets, make training on a single device impractical or even impossible. Hybrid parallelism enables the division of the model across multiple devices to manage memory constraints while simultaneously distributing the dataset to process vast amounts of data efficiently. This dual capability ensures that training systems can handle the computational and memory demands of the largest models and datasets without compromise.

Another critical advantage lies in hardware utilization. In many distributed training systems, inefficiencies can arise when devices sit idle during different stages of computation or synchronization. Hybrid parallelism mitigates this issue by ensuring that all devices are actively engaged. Whether a device is computing forward passes through its portion of the model or processing data batches, hybrid strategies maximize resource usage, leading to faster training times and improved throughput.

Flexibility is another hallmark of hybrid parallelism. Machine learning models vary widely in architecture and computational demands. For instance, convolutional neural networks prioritize spatial data processing, while transformers require intensive operations like matrix multiplications in attention mechanisms. Hybrid parallelism adapts to these diverse needs by allowing practitioners to apply model and data parallelism selectively. This adaptability ensures that hybrid approaches can be tailored to the specific requirements of a given model, making it a versatile solution for diverse training scenarios.

Moreover, hybrid parallelism reduces communication bottlenecks, a common issue in distributed systems. By striking a balance between distributing model computations and spreading data processing, hybrid strategies minimize the amount of inter-device communication required during training. This efficient coordination not only speeds up the training process but also enables the effective use of large-scale distributed systems where network latency might otherwise limit performance.

Finally, hybrid parallelism supports the ambitious scale of modern AI research and development. It provides a framework for leveraging cutting-edge hardware infrastructures, including clusters of GPUs or TPUs, to train models that push the boundaries of what's possible. Without hybrid parallelism, many of the breakthroughs in AI, including large language models and advanced vision systems, would remain unattainable due to resource limitations.

By enabling scalability, maximizing hardware efficiency, and offering flexibility, hybrid parallelism has become an essential strategy for training the most complex machine learning systems. It is not just a solution to today's challenges but also a foundation for the future of AI, where models and datasets will continue to grow in complexity and size.

8.6.3.3 Challenges

While hybrid parallelism provides a robust framework for scaling machine learning training, it also introduces complexities that require careful consideration. These challenges stem from the intricate coordination needed to integrate both model and data parallelism effectively. Understanding these obstacles is crucial for designing efficient hybrid systems and avoiding potential bottlenecks.

One of the primary challenges of hybrid parallelism is communication overhead. Both model and data parallelism involve significant inter-device communication. In model parallelism, devices must exchange intermediate outputs and gradients to maintain the sequential flow of computation. In data parallelism, gradients computed on separate data subsets must be synchronized across devices. Hybrid parallelism compounds these demands, as it requires efficient communication for both processes simultaneously. If not managed properly, the resulting overhead can negate the benefits of parallelization, particularly in large-scale systems with slower interconnects or high network latency.

Another critical challenge is the complexity of implementation. Hybrid parallelism demands a nuanced understanding of both model and data parallelism techniques, as well as the underlying hardware and software infrastructure. Designing efficient hybrid strategies involves making decisions about how to partition the model, how to distribute data, and how to synchronize computations across devices. This process often requires extensive experimentation and optimization, particularly for custom architectures or non-standard hardware setups. While modern frameworks like PyTorch and TensorFlow provide tools for distributed training, implementing hybrid parallelism at scale still requires significant engineering expertise.

Workload balancing also presents a challenge in hybrid parallelism. In a distributed system, not all devices may have equal computational capacity. Some devices may process data or compute gradients faster than others, leading to inefficiencies as faster devices wait for slower ones to complete their tasks. Additionally, certain model layers or operations may require more resources than others, creating imbalances in computational load. Managing this disparity requires careful tuning of partitioning strategies and the use of dynamic workload distribution techniques.

Memory constraints remain a concern, even in hybrid setups. While model parallelism addresses the issue of fitting large models into device memory, the additional memory requirements for data parallelism, such as storing multiple data batches and gradient buffers, can still exceed available capacity. This is especially true for models with extremely large intermediate computations, such as transformers with high-dimensional attention mechanisms. Balancing memory usage across devices is essential to prevent resource exhaustion during training.

Lastly, hybrid parallelism poses challenges related to fault tolerance and debugging. Distributed systems are inherently more prone to hardware failures and synchronization errors. Debugging issues in hybrid setups can be significantly more complex than in standalone model or data parallelism systems, as errors may arise from interactions between the two approaches. Ensuring robust fault-tolerance mechanisms and designing tools for monitoring and debugging distributed systems are essential for maintaining reliability.

Despite these challenges, hybrid parallelism remains an indispensable strategy for training state-of-the-art machine learning models. By addressing these obstacles through optimized communication protocols, intelligent partitioning strategies, and robust fault-tolerance systems, practitioners can unlock the full potential of hybrid parallelism and drive innovation in AI research and applications.

8.6.4 Comparison

The features of data parallelism, model parallelism, and hybrid parallelism are summarized in Table 8.6. This comparison highlights their respective focuses, memory requirements, communication overheads, scalability, implementation complexity, and ideal use cases. By examining these factors, practitioners can determine the most suitable approach for their training needs.

Table 8.6: Comparison of data parallelism, model parallelism, and hybrid parallelism across key aspects.

Aspect	Data Parallelism	Model Parallelism	Hybrid Parallelism
Focus	Distributes dataset across devices, each with a full model copy	Distributes the model across devices, each handling a portion of the model	Combines model and data parallelism for balanced scalability
Memory Requirement per Device	High (entire model on each device)	Low (model split across devices)	Moderate (splits model and dataset across devices)
Communication Overhead	Moderate to High (gradient synchronization across devices)	High (communication for intermediate activations and gradients)	Very High (requires synchronization for both model and data)
Scalability	Good for large datasets with moderate model sizes	Good for very large models with smaller datasets	Excellent for extremely large models and datasets
Implementation Complexity	Low to Moderate (relatively straightforward with existing tools)	Moderate to High (requires careful partitioning and coordination)	High (complex integration of model and data parallelism)
Ideal Use Case	Large datasets where model fits within a single device	Extremely large models that exceed single-device memory limits	Training massive models on vast datasets in large-scale systems

Figure 8.19 provides a general guideline for selecting parallelism strategies in distributed training systems. While the chart offers a structured decision-making process based on model size, dataset size, and scaling constraints, it is intentionally simplified. Real-world scenarios often involve additional complexities such as hardware heterogeneity, communication bandwidth, and workload imbalance, which may influence the choice of parallelism techniques. The chart

is best viewed as a foundational tool for understanding the trade-offs and decision points in parallelism strategy selection. Practitioners should consider this guideline as a starting point and adapt it to the specific requirements and constraints of their systems to achieve optimal performance.

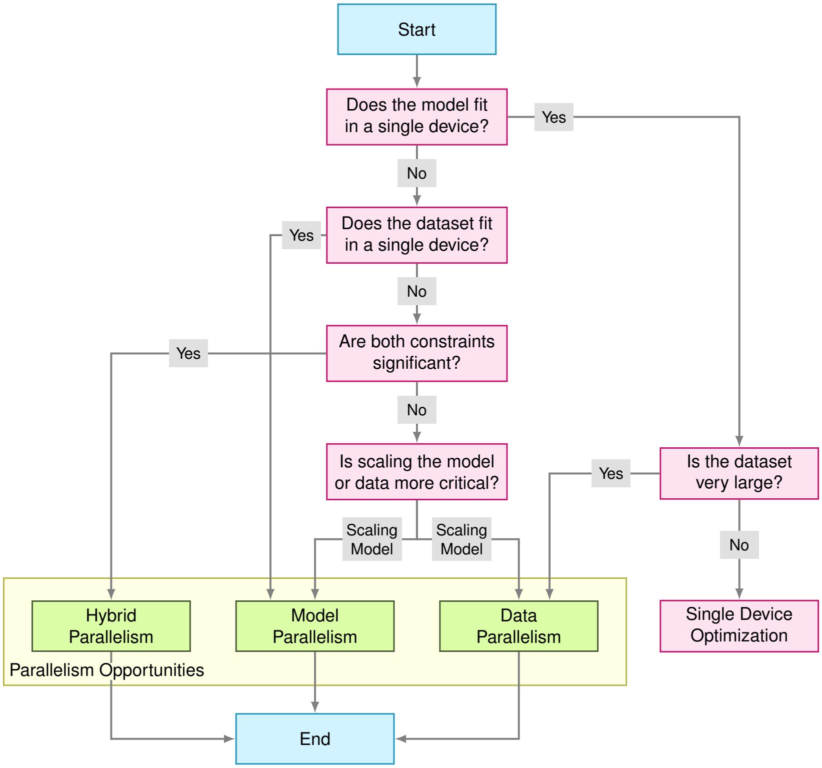

Start
Does the model fit in a single device?
Yes
No
Does the dataset fit in a single device?
Yes
No
Are both constraints significant?
Yes
No
Is scaling the model or data more critical?
Scaling Model
Scaling Model
Is the dataset very large?
Yes
No
Hybrid Parallelism
Model Parallelism
Data Parallelism
Single Device Optimization
Parallelism Opportunities
End

Figure 8.19: Decision flowchart for selecting parallelism strategies in distributed training.

8.7 Optimization Techniques

Efficient training of machine learning models relies on identifying and addressing the factors that limit performance and scalability. This section explores a range of optimization techniques designed to improve the efficiency of training systems. By targeting specific bottlenecks, optimizing hardware and software interactions, and employing scalable training strategies, these methods enable practitioners to build systems that effectively utilize resources while minimizing training time.

8.7.1 Identifying Bottlenecks

Effective optimization of training systems requires a systematic approach to identifying and addressing performance bottlenecks. Bottlenecks can arise at various levels, including computation, memory, and data handling, and they directly impact the efficiency and scalability of the training process.

Computational bottlenecks can significantly impact training efficiency. One common bottleneck occurs when computational resources, such as GPUs or TPUs, are underutilized. This can happen due to imbalanced workloads or inefficient parallelization strategies. For example, if one device completes its assigned computation faster than others, it remains idle while waiting for the slower devices to catch up. Such inefficiencies reduce the overall training throughput.

Memory-related bottlenecks are particularly challenging when dealing with large models. Insufficient memory can lead to frequent swapping of data between device memory and slower storage, significantly slowing down the training process. In some cases, the memory required to store intermediate activations during the forward and backward passes can exceed the available capacity, forcing the system to employ techniques such as gradient checkpointing, which trade off computational efficiency for memory savings.

Data handling bottlenecks can severely limit the utilization of computational resources. Training systems often rely on a continuous supply of data to keep computational resources fully utilized. If data loading and preprocessing are not optimized, computational devices may sit idle while waiting for new batches of data to arrive. This issue is particularly prevalent when training on large datasets stored on networked file systems or remote storage solutions. As illustrated in Figure 8.20, profiling traces can reveal cases where the GPU remains underutilized due to slow data loading, highlighting the importance of efficient input pipelines.

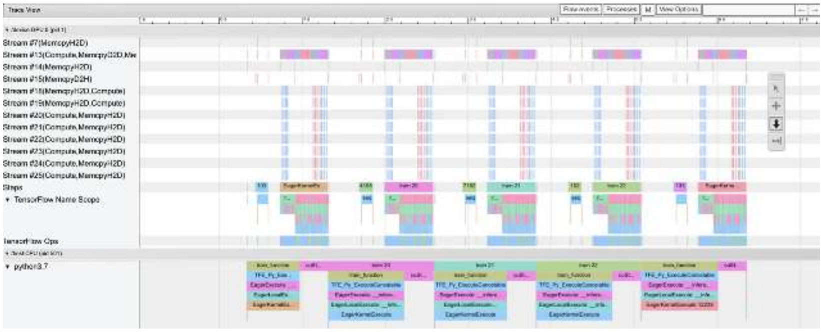

Trace View
Flow events
Processes
M
View Options
Stream #7(MemcpyH2D)
Stream #13(Compute,MemcpyD2D,Me
Stream #14(MemcpyH2D)
Stream #15(MemcpyD2H)
Stream #18(MemcpyH2D,Compute)
Stream #19(MemcpyH2D,Compute)
Stream #20(Compute,MemcpyH2D)
Stream #21(Compute,MemcpyH2D)
Stream #22(Compute,MemcpyH2D)
Stream #23(Compute,MemcpyH2D)
Stream #24(Compute,MemcpyH2D)
Stream #25(Compute,MemcpyH2D)
Steps
TensorFlow Name Scope
TensorFlow Ops
python3.7

Figure 8.20: Example TensorFlow profiling trace showing low utilization. We observe that this workload is bounded by the dataloader, as the GPU sits largely unutilized waiting for work.

Identifying these bottlenecks typically involves using profiling tools to analyze the performance of the training system. Tools integrated into machine learning frameworks, such as PyTorch's `torch.profiler` or TensorFlow's `tf.data` analysis utilities, can provide detailed insights into where time and resources are being spent during training. By pinpointing the specific stages or operations that are causing delays, practitioners can design targeted optimizations to address these issues effectively.

8.7.2 System-Level Optimizations

After identifying the bottlenecks in a training system, the next step is to implement optimizations at the system level. These optimizations target the underlying hardware, data flow, and resource allocation to improve overall performance and scalability.

One essential technique is profiling training workloads. Profiling involves collecting detailed metrics about the system's performance during training, such as computation times, memory usage, and communication overhead. These metrics help reveal inefficiencies, such as imbalanced resource usage or excessive time spent in specific stages of the training pipeline. Profiling tools such as NVIDIA Nsight Systems or TensorFlow Profiler can provide actionable insights, enabling developers to make informed adjustments to their training configurations.

Leveraging hardware-specific features is another critical aspect of system-level optimization. Modern accelerators, such as GPUs and TPUs, include specialized capabilities that can significantly enhance performance when utilized effectively. For instance, mixed precision training, which uses lower-precision floating-point formats like FP16 or bfloat16[13] for computations, can dramatically reduce memory usage and improve throughput without sacrificing model accuracy. Similarly, tensor cores in NVIDIA GPUs are designed to accelerate matrix operations, a common computational workload in deep learning, making them ideal for optimizing forward and backward passes.

Data pipeline optimization is also an important consideration at the system level. Ensuring that data is loaded, preprocessed, and delivered to the training devices efficiently can eliminate potential bottlenecks caused by slow data delivery. Techniques such as caching frequently used data, prefetching batches to overlap computation and data loading, and using efficient data storage formats like TFRecord or RecordIO can help maintain a steady flow of data to computational devices.

8.7.3 Software-Level Optimizations

In addition to system-level adjustments, software-level optimizations focus on improving the efficiency of training algorithms and their implementation within machine learning frameworks.

One effective software-level optimization is the use of fused kernels. In traditional implementations, operations like matrix multiplications, activation functions, and gradient calculations are often executed as separate steps. Fused kernels combine these operations into a single optimized routine, reducing the overhead associated with launching multiple operations and improving cache utilization. Many frameworks, such as PyTorch and TensorFlow, automatically apply kernel fusion where possible, but developers can further optimize custom operations by explicitly using libraries like cuBLAS or cuDNN.

Dynamic graph execution is another powerful technique for software-level optimization. In frameworks that support dynamic computation graphs, such as PyTorch, the graph of operations is constructed on-the-fly during each training iteration. This flexibility allows for fine-grained optimizations based on the specific inputs and outputs of a given iteration. Dynamic graphs also enable more efficient handling of variable-length sequences, such as those encountered in natural language processing tasks.

[13] Google's bfloat16 format retains FP32's dynamic range while reducing precision, making it highly effective for deep learning training on TPUs.

Gradient accumulation is an additional strategy that can be implemented at the software level to address memory constraints. Instead of updating model parameters after every batch, gradient accumulation allows the system to compute gradients over multiple smaller batches and update parameters only after aggregating them. This approach effectively increases the batch size without requiring additional memory, enabling training on larger datasets or models.

8.7.4 Scaling Techniques

Scaling techniques aim to extend the capabilities of training systems to handle larger datasets and models by optimizing the training configuration and resource allocation.

One common scaling technique is batch size scaling. Increasing the batch size can reduce the number of synchronization steps required during training, as fewer updates are needed to process the same amount of data. However, larger batch sizes may introduce challenges, such as slower convergence or reduced generalization. Techniques like learning rate scaling and warmup schedules can help mitigate these issues, ensuring stable and effective training even with large batches.

Layer-freezing strategies provide another method for scaling training systems efficiently. In many scenarios, particularly in transfer learning, the lower layers of a model capture general features and do not need frequent updates. By freezing these layers and allowing only the upper layers to train, memory and computational resources can be conserved, enabling the system to focus its efforts on fine-tuning the most critical parts of the model.

8.8 Specialized Hardware Training

The evolution of specialized machine learning hardware represents a critical development in addressing the computational demands of modern training systems. Each hardware architecture, such as GPUs, TPUs, FPGAs, and ASICs, embodies distinct design philosophies and engineering trade-offs that optimize for specific aspects of the training process. These specialized processors have fundamentally altered the scalability and efficiency constraints of machine learning systems, enabling breakthroughs in model complexity and training speed. We briefly examine the architectural principles, performance characteristics, and practical applications of each hardware type, highlighting their indispensable role in shaping the future capabilities of machine learning training systems.

8.8.1 GPUs

Machine learning training systems demand immense computational power to process large datasets, perform gradient computations, and update model parameters efficiently. GPUs have emerged as a critical technology to meet these requirements (Figure 8.21), primarily due to their highly parallelized architecture and ability to execute the dense linear algebra operations central to neural network training (Dally, Keckler, and Kirk 2021).

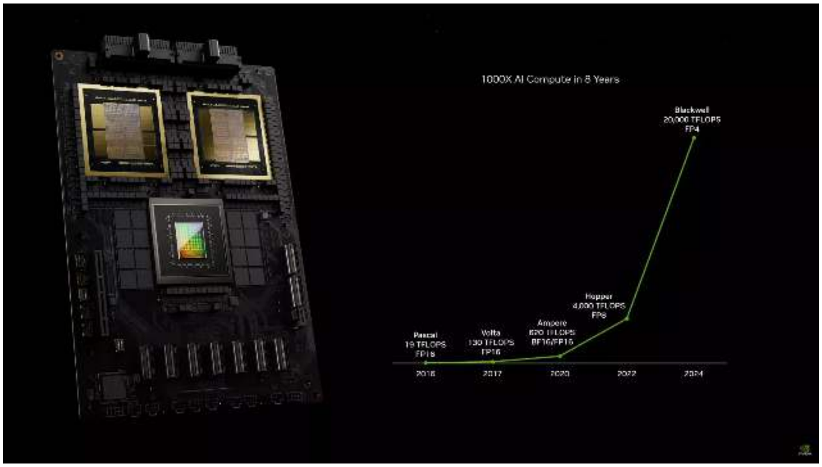


Figure 8.21: GPU design has dramatically accelerated AI training, enabling breakthroughs in large-scale models like GPT-3.

From the perspective of training pipeline architecture, GPUs address several key bottlenecks. The large number of cores in GPUs allows for simultaneous processing of thousands of matrix multiplications, accelerating the forward and backward passes of training. In systems where data throughput limits GPU utilization, prefetching and caching mechanisms help maintain a steady flow of data. These optimizations, previously discussed in training pipeline design, are critical to unlocking the full potential of GPUs (D. A. Patterson and Hennessy 2021b).

In distributed training systems, GPUs enable scalable strategies such as data parallelism and model parallelism. NVIDIA's ecosystem, including tools like NCCL for multi-GPU communication, facilitates efficient parameter synchronization, a frequent challenge in large-scale setups. For example, in training large models like GPT-3, GPUs were used in tandem with distributed frameworks to split computations across thousands of devices while addressing memory and compute scaling issues (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, Dhariwal, et al. 2020).

Hardware-specific features further enhance GPU performance. NVIDIA's tensor cores, for instance, are optimized for mixed-precision training, which reduces memory usage while maintaining numerical stability (Micikevicius et al. 2017b). This directly addresses memory constraints, a common bottleneck in training massive models. Combined with software-level optimizations like fused kernels, GPUs deliver substantial speedups in both single-device and multi-device configurations.

A case study that exemplifies the role of GPUs in machine learning training is OpenAI's use of NVIDIA hardware for large language models. Training GPT-3, with its 175 billion parameters, required distributed processing across thousands of V100 GPUs. The combination of GPU-optimized frameworks, advanced communication protocols, and hardware features enabled OpenAI to achieve this ambitious scale efficiently (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, Dhariwal, et al. 2020).

Despite their advantages, GPUs are not without challenges. Effective utilization of GPUs demands careful attention to workload balancing and inter-device communication. Training systems must also consider the cost implications, as GPUs are resource-intensive and require optimized data centers to operate at scale. However, with innovations like NVLink and CUDA-X libraries, these challenges are continually being addressed.

In conclusion, GPUs are indispensable for modern machine learning training systems due to their versatility, scalability, and integration with advanced software frameworks. By addressing key bottlenecks in computation, memory, and distribution, GPUs play a foundational role in enabling the large-scale training pipelines discussed throughout this chapter.

8.8.2 TPUs

Tensor Processing Units (TPUs) and other custom accelerators have been purpose-built to address the unique challenges of large-scale machine learning training. Unlike GPUs, which are versatile and serve a wide range of applications, TPUs are specifically optimized for the computational patterns found in deep learning, such as matrix multiplications and convolutional operations (Jouppi, Young, et al. 2017c). These devices mitigate training bottlenecks by offering high throughput, specialized memory handling, and tight integration with machine learning frameworks.

As illustrated in Figure 8.22, TPUs have undergone significant architectural evolution, with each generation introducing enhancements tailored for increasingly demanding AI workloads. The first-generation TPU, introduced in 2015, was designed for internal inference acceleration. Subsequent iterations have focused on large-scale distributed training, memory optimizations, and efficiency improvements, culminating in the most recent Trillium architecture. These advancements illustrate how domain-specific accelerators continue to push the boundaries of AI performance and efficiency.

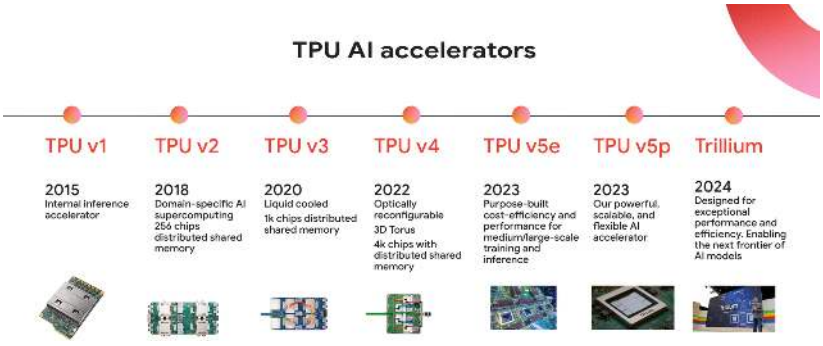

TPU AI accelerators
TPU v1
2015
Internal inference accelerator
TPU v2
2018
Domain-specific AI supercomputing
256 chips distributed shared memory
TPU v3
2020
Liquid cooled
1k chips distributed shared memory
TPU v4
2022
Optically reconfigurable
3D Torus
4k chips with distributed shared memory
TPU v5e
2023
Purpose-built cost-efficiency and performance for medium/large-scale training and inference
TPU v5p
2023
Our powerful, scalable, and flexible AI accelerator
Trillium
2024
Designed for exceptional performance and efficiency. Enabling the next frontier of AI models

Figure 8.22: Tensor Processing Units (TPUs), a single, specific purpose chip desigend for accelerated AI.

Machine learning frameworks can achieve substantial gains in training efficiency through purpose-built AI accelerators such as TPUs. However, maximizing these benefits requires careful attention to hardware-aware optimizations, including memory layout, dataflow orchestration, and computational efficiency.

Google developed TPUs with a primary goal: to accelerate machine learning workloads at scale while reducing the energy and infrastructure costs associated with traditional hardware. Their architecture is optimized for tasks that benefit from batch processing, making them particularly effective in distributed training systems where large datasets are split across multiple devices. A key feature of TPUs is their systolic array architecture, which performs efficient matrix multiplications by streaming data through a network of processing elements. This design minimizes data movement overhead, reducing latency and energy consumption—critical factors for training large-scale models like transformers (Jouppi, Young, et al. 2017c).

From the perspective of training pipeline optimization, TPUs simplify integration with data pipelines in the TensorFlow ecosystem. Features such as the TPU runtime and TensorFlow's `tf.data` API enable seamless preprocessing, caching, and batching of data to feed the accelerators efficiently (Martín Abadi, Agarwal, et al. 2016). Additionally, TPUs are designed to work in pods—clusters of interconnected TPU devices that allow for massive parallelism. In such setups, TPU pods enable hybrid parallelism strategies by combining data parallelism across devices with model parallelism within devices, addressing memory and compute constraints simultaneously.

TPUs have been instrumental in training large-scale models, such as BERT and T5. For example, Google's use of TPUs to train BERT demonstrates their ability to handle both the memory-intensive requirements of large transformer models and the synchronization challenges of distributed setups (Devlin et al. 2018). By splitting the model across TPU cores and optimizing communication patterns, Google achieved state-of-the-art results while significantly reducing training time compared to traditional hardware.

Beyond TPUs, custom accelerators such as AWS Trainium and Intel Gaudi chips are also gaining traction in the machine learning ecosystem. These devices are designed to compete with TPUs by offering similar performance benefits while catering to diverse cloud and on-premise environments. For example, AWS Trainium provides deep integration with the AWS ecosystem, allowing users to seamlessly scale their training pipelines with services like Amazon SageMaker.

While TPUs and custom accelerators excel in throughput and energy efficiency, their specialized nature introduces limitations. TPUs, for example, are tightly coupled with Google's ecosystem, making them less accessible to practitioners using alternative frameworks. Similarly, the high upfront investment required for TPU pods may deter smaller organizations or those with limited budgets. Despite these challenges, the performance gains offered by custom accelerators make them a compelling choice for large-scale training tasks.

In summary, TPUs and custom accelerators address many of the key challenges in machine learning training systems, from handling massive datasets to optimizing distributed training. Their unique architectures and deep integration with specific ecosystems make them powerful tools for organizations seeking to scale their training workflows. As machine learning models and datasets continue to grow, these accelerators are likely to play an increasingly central role in shaping the future of AI training.

8.8.3 FPGAs

Field-Programmable Gate Arrays (FPGAs) are versatile hardware solutions that allow developers to tailor their architecture for specific machine learning workloads. Unlike GPUs or TPUs, which are designed with fixed architectures, FPGAs can be reconfigured dynamically, offering a unique level of flexibility. This adaptability makes them particularly valuable for applications that require customized optimizations, low-latency processing, or experimentation with novel algorithms.

Microsoft had been exploring the use of FPGAs for a while, as seen in Figure 8.23, with one prominent example being Project Brainwave. This initiative leverages FPGAs to accelerate machine learning workloads in the Azure cloud. Microsoft chose FPGAs for their ability to provide low-latency inference (not training) while maintaining high throughput. This approach is especially beneficial in scenarios where real-time predictions are critical, such as search engine queries or language translation services. By integrating FPGAs directly into their data center network, Microsoft has achieved significant performance gains while minimizing power consumption.

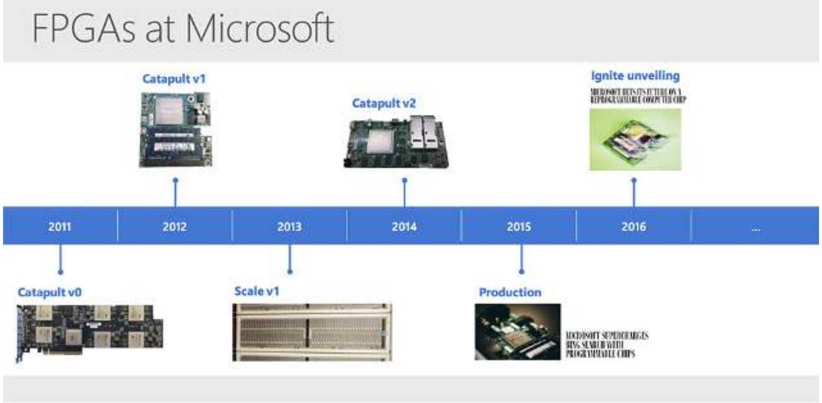

FPGAs at Microsoft
Catapult v1
Catapult v2
Ignite unveiling
MICROSOFT BETS ITS FUTURE ON A REPROGRAMMABLE COMPUTER CHIP
2011
2012
2013
2014
2015
2016
...
Catapult v0
Scale v1
Production
MICROSOFT SUPERCHARGES BING SEARCH WITH PROGRAMMABLE CHIPS

Figure 8.23: Microsoft's FPGA advancements through Project Catapult and Project Brainwave highlight their focus on accelerating AI and other cloud workloads with reconfigurable hardware.

From a training perspective, FPGAs offer unique advantages in optimizing training pipelines. Their reconfigurability allows them to implement custom dataflow architectures tailored to specific model requirements. For instance, data preprocessing and augmentation steps, which can often become bottlenecks in GPU-based systems, can be offloaded to FPGAs, freeing up GPUs for core training tasks. Additionally, FPGAs can be programmed to perform operations such as sparse matrix multiplications, which are common in recommendation systems and graph-based models but are less efficient on traditional accelerators (Putnam et al. 2014).

In distributed training systems, FPGAs provide fine-grained control over communication patterns. This control allows developers to optimize inter-device communication and memory access, addressing challenges such as parameter synchronization overheads. For example, FPGAs can be configured to implement custom all-reduce algorithms for gradient aggregation, reducing latency compared to general-purpose hardware.

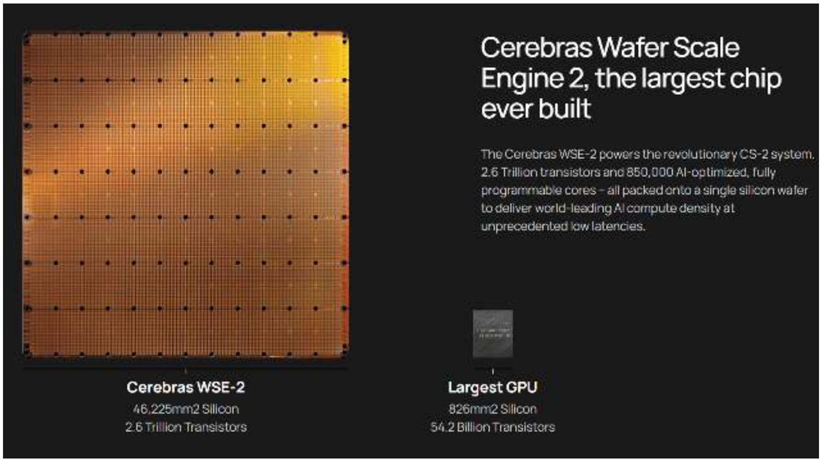
Despite their benefits, FPGAs come with challenges. Programming FPGAs requires expertise in hardware description languages (HDLs) like Verilog or VHDL, which can be a barrier for many machine learning practitioners. To address this, frameworks like Xilinx's Vitis AI and Intel's OpenVINO have simplified FPGA programming by providing tools and libraries tailored for AI workloads. However, the learning curve remains steep compared to the well-established ecosystems of GPUs and TPUs.

Microsoft's use of FPGAs highlights their potential to integrate seamlessly into existing machine learning workflows. By incorporating FPGAs into Azure, Microsoft has demonstrated how these devices can complement other accelerators, optimizing end-to-end pipelines for both training and inference. This hybrid approach leverages the strengths of FPGAs for specific tasks while relying on GPUs or CPUs for others, creating a balanced and efficient system.

In summary, FPGAs offer a compelling solution for machine learning training systems that require customization, low latency, or novel optimizations. While their adoption may be limited by programming complexity, advancements in tooling and real-world implementations like Microsoft's Project Brainwave demonstrate their growing relevance in the AI hardware ecosystem.

8.8.4 ASICs

Application-Specific Integrated Circuits (ASICs) represent a class of hardware designed for specific tasks, offering unparalleled efficiency and performance by eschewing the general-purpose flexibility of GPUs or FPGAs. Among the most innovative examples of ASICs for machine learning training is the Cerebras Wafer-Scale Engine (WSE), as shown in Figure 8.24, which stands apart for its unique approach to addressing the computational and memory challenges of training massive machine learning models.



Cerebras Wafer Scale Engine 2, the largest chip ever built

The Cerebras WSE-2 powers the revolutionary CS-2 system. 2.6 Trillion transistors and 850,000 AI-optimized, fully programmable cores – all packed onto a single silicon wafer to deliver world-leading AI compute density at unprecedented low latencies.

Figure 8.24: Cerebras Wafer Scale Engine (WSE) 2 is the largest AI chip ever built with nearly a cores.

The Cerebras WSE is unlike traditional chips in that it is a single wafer-scale processor, spanning the entire silicon wafer rather than being cut into smaller chips. This architecture enables Cerebras to pack 2.6 trillion transistors and 850,000 cores onto a single device. These cores are connected via a high-bandwidth, low-latency interconnect, allowing data to move across the chip without the bottlenecks associated with external communication between discrete GPUs or TPUs (Feldman et al. 2020).

From a machine learning training perspective, the WSE addresses several critical bottlenecks:

1. **Data Movement**: In traditional distributed systems, significant time is spent transferring data between devices. The WSE eliminates this by keeping all computations and memory on a single wafer, drastically reducing communication overhead.
2. **Memory Bandwidth**: The WSE integrates 40 GB of high-speed on-chip memory directly adjacent to its processing cores. This proximity allows for near-instantaneous access to data, overcoming the latency challenges that GPUs often face when accessing off-chip memory.
3. **Scalability**: While traditional distributed systems rely on complex software frameworks to manage multiple devices, the WSE simplifies scaling by consolidating all resources into one massive chip. This design is particularly well-suited for training large language models and other deep learning architectures that require significant parallelism.

A key example of Cerebras' impact is its application in natural language processing. Organizations using the WSE have demonstrated substantial speedups in training transformer models, which are notoriously compute-intensive due to their reliance on attention mechanisms. By leveraging the chip's massive parallelism and memory bandwidth, training times for models like BERT have been significantly reduced compared to GPU-based systems (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, Dhariwal, et al. 2020).

However, the Cerebras WSE also comes with limitations. Its single-chip design is optimized for specific use cases, such as dense matrix computations in deep learning, but may not be as versatile as multi-purpose hardware like GPUs or FPGAs. Additionally, the cost of acquiring and integrating such a specialized device can be prohibitive for smaller organizations or those with diverse workloads.

Cerebras' strategy of targeting the largest models aligns with the trends discussed earlier in this chapter, such as the growing emphasis on scaling techniques and hybrid parallelism strategies. The WSE's unique design addresses challenges like memory bottlenecks and inter-device communication overhead, making it a pioneering solution for next-generation AI workloads.

In conclusion, the Cerebras Wafer-Scale Engine exemplifies how ASICs can push the boundaries of what is possible in machine learning training. By addressing fundamental bottlenecks in computation and data movement, the WSE offers a glimpse into the future of specialized hardware for AI, where the integration of highly optimized, task-specific architectures unlocks unprecedented performance.

8.9 Conclusion

AI training systems are built upon a foundation of mathematical principles, computational strategies, and architectural considerations. The exploration of neural network computation has shown how core operations, activation functions, and optimization algorithms come together to enable efficient model training, while also emphasizing the trade-offs that must be balanced between memory, computation, and performance.

The design of training pipelines incorporates key components such as data flows, forward and backward passes, and memory management. Understanding these elements in conjunction with hardware execution patterns is essential for achieving efficient and scalable training processes. Strategies like parameter updates, prefetching, and gradient accumulation further enhance the effectiveness of training by optimizing resource utilization and reducing computational bottlenecks.

Distributed training systems, including data parallelism, model parallelism, and hybrid approaches, are topics that we examined as solutions for scaling AI training to larger datasets and models. Each approach comes with its own benefits and challenges, highlighting the need for careful consideration of system requirements and resource constraints.

Altogether, the combination of theoretical foundations and practical implementations forms a cohesive framework for addressing the complexities of AI training. By leveraging this knowledge, it is possible to design robust, efficient systems capable of meeting the demands of modern machine learning applications.

8.10 Resources

ℹ Slides

- *Coming soon.*

❗ Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 9

Efficient AI

Figure 9.1: *DALL·E 3 Prompt: A conceptual illustration depicting efficiency in artificial intelligence using a shipyard analogy. The scene shows a bustling shipyard where containers represent bits or bytes of data. These containers are being moved around efficiently by cranes and vehicles, symbolizing the streamlined and rapid information processing in AI systems. The shipyard is meticulously organized, illustrating the concept of optimal performance within the constraints of limited resources. In the background, ships are docked, representing different platforms and scenarios where AI is applied. The atmosphere should convey advanced technology with an underlying theme of sustainability and wide applicability.*

Purpose

What principles guide the efficient design of machine learning systems, and why is understanding the interdependence of key resources essential?

Machine learning systems are shaped by the complex interplay among data, models, and computing resources. Decisions on efficiency in one dimension often have ripple effects in the others, presenting both opportunities for synergy and inevitable trade-offs. Understanding these individual components and their interdependencies exposes not only how systems can be optimized but also why these optimizations are crucial for achieving scalability, sustainability, and real-world applicability. The relationship between data, model, and computing efficiency forms the basis for designing machine learning systems that maximize capabilities while working within resource limitations. Each efficiency decision represents a balance between performance and practicality, underscoring the significance of a holistic approach to system design. Exploring these relationships equips us with the strategies necessary to navigate the intricacies of developing efficient, impactful AI solutions.

Learning Objectives

- Define the principles of algorithmic, compute and data efficiency in AI systems.
- Identify and analyze trade-offs between algorithmic, compute, and data efficiency in system design.
- Apply strategies for achieving efficiency across diverse deployment contexts, such as edge, cloud, and Tiny ML applications.
- Examine the historical evolution and emerging trends in machine learning efficiency.
- Evaluate the broader ethical and environmental implications of efficient AI system design.

9.1 Overview

Machine learning systems have become ubiquitous, permeating nearly every aspect of modern life. As these systems grow in complexity and scale, they must operate effectively across a wide range of deployments and scenarios. This necessitates careful consideration of factors such as processing speed, memory usage, and power consumption to ensure that models can handle large workloads, operate on energy-constrained devices, and remain cost-effective.

Achieving this balance involves navigating trade-offs. For instance, in autonomous vehicles, reducing a model's size to fit the low-power constraints of an edge device in a car might slightly decrease accuracy, but it ensures real-time processing and decision-making. Conversely, a cloud-based system can afford higher model complexity for improved accuracy, though this often comes at the cost of increased latency and energy consumption. In the medical field, deploying machine learning models on portable devices for diagnostics requires efficient models that can operate with limited computational resources and power, ensuring accessibility in remote or resource-constrained areas. Conversely, hospital-based systems can leverage more powerful hardware to run complex models for detailed analysis, albeit with higher energy demands.

Understanding and managing these trade-offs is crucial for designing machine learning systems that meet diverse application needs within real-world constraints. The implications of these design choices extend beyond performance and cost. Efficient systems can be deployed across diverse environments, from cloud infrastructures to edge devices, enhancing accessibility and adoption. Additionally, they help reduce the environmental impact of machine learning workloads by lowering energy consumption and carbon emissions, aligning technological progress with ethical and ecological responsibilities.

This chapter focuses on the 'why' and 'how' of efficiency in machine learning systems. By establishing the foundational principles of efficient AI and exploring strategies to achieve it, this chapter sets the stage for deeper discussions on topics such as scaling, optimization, deployment, and sustainability in later chapters.

9.2 AI Scaling Laws

The advancement of machine learning systems has been characterized by a consistent trend: the augmentation of model scale, encompassing parameters, training data, and computational resources, typically results in enhanced performance. This observation, which was discovered empirically, has driven significant progress across domains such as natural language processing, computer vision, and speech recognition, where larger models trained on extensive datasets have consistently achieved state-of-the-art results.

However, the pursuit of performance gains through scaling incurs substantial resource demands, raising critical inquiries regarding the efficiency and sustainability of such practices. Specifically, questions arise concerning the incremental computational resources required for marginal improvements in accuracy, the scalability of data requirements as task complexity increases, and the point at which diminishing returns render further scaling economically or practically infeasible.

To address these concerns, researchers have developed scaling laws—empirical relationships that quantify the correlation between model performance and training resources. These laws provide a formal framework for analyzing the trade-offs inherent in scaling and elucidate the increasing importance of efficiency as systems expand in size and complexity.

This section introduces the concept of scaling laws, delineates their manifestation across model, compute, and data dimensions, and examines their implications for system design. By doing so, it establishes a foundation for understanding the limitations of brute-force scaling and underscores the necessity of efficient methodologies that balance performance with practical resource constraints.

9.2.1 Fundamental Principles

To comprehend the intricacies of efficiency in large-scale machine learning systems, it is imperative to establish the fundamental principles that govern their performance. These principles are encapsulated in scaling laws, which describe the empirical relationships between model performance and the allocation of resources, including model size, dataset size, and computational capacity. While initially popularized within the context of large language models, the implications of scaling laws extend across diverse domains of machine learning.

The genesis of scaling laws in machine learning is intrinsically linked to the advent of deep learning and the proliferation of large-scale models. In the 2010s, researchers observed a consistent trend: augmenting the size of neural networks resulted in notable performance enhancements (Hestness et al. 2017), particularly in complex tasks such as image recognition and natural language processing. This observation, termed the 'scaling hypothesis,' posited that larger models possess an increased capacity to capture intricate data patterns, thereby facilitating improved accuracy and generalization. Consequently, the field witnessed a surge in models with millions or billions of parameters, trained on extensive datasets to attain state-of-the-art results. However, this trend also precipitated concerns regarding the sustainability and efficiency of scaling, necessitating a rigorous examination of the associated trade-offs.

Scaling laws provide a quantitative framework for analyzing these trade-offs. They elucidate how model performance, training time, and resource consumption vary with scale, enabling researchers to identify optimal strategies for developing high-performing, resource-efficient systems. These laws have become indispensable tools for guiding the design of contemporary machine learning architectures, ensuring that advancements in scale are harmonized with broader objectives of efficiency and sustainability.

Scaling laws reveal that model performance exhibits predictable patterns as resources are augmented. For example, power-law scaling,[0] a common phenomenon in deep learning, posits that performance improves as a power function of model size, dataset size, or computational resources. This relationship can be mathematically expressed as:

$$\text{Performance} \propto \text{Resource}^{-\alpha}$$

where α denotes a scaling exponent that varies based on the task and model architecture. This expression indicates that increasing model size, dataset size, or computational resources leads to predictable enhancements in performance, adhering to a power-law relationship.

Empirical studies of large language models (LLMs) further elucidate the interplay between these factors, parameters, data, and computational resources, under fixed resource constraints. As illustrated in Figure 9.2, for a given computational budget in language model training, there exists an optimal allocation between model size and dataset size (measured in tokens) that minimizes training loss. The left panel depicts 'IsoFLOP curves,' where each curve corresponds to a constant number of floating-point operations (FLOPs) during transformer training. Each valley in these curves signifies the most efficient model size for a given computational level when training autoregressive language models. The center and right panels demonstrate how the optimal number of parameters and tokens scales predictably with increasing computational budgets in language model training, highlighting the necessity for coordinated scaling to maximize resource utilization in large language models.

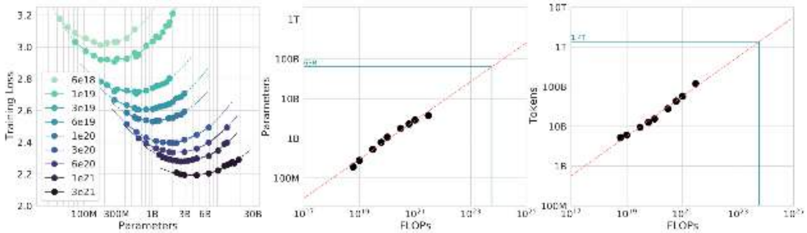


Figure 9.2: The left panel shows training loss as a function of model size for fixed compute budgets, revealing that there exists an optimal parameter count for each compute level. The central and right panels depict how the optimal number of model parameters and training tokens scale with available FLOPs. These empirical curves illustrate the need to balance model size and data volume when scaling under resource constraints. Source: (Hoffmann et al. 2022).

For instance, in computer vision tasks, doubling the size of convolutional neural networks typically yields consistent accuracy gains, provided that pro-

[0] Power-law relationship: A mathematical relationship where one quantity varies as a power of another. In ML scaling laws, performance improvements are proportional to resource increases raised to some power, showing diminishing returns as resources scale up.

portional increases in training data are supplied. Similarly, language models exhibit analogous patterns, with studies of models such as GPT-3 demonstrating that performance scales predictably with both model parameters and training data volume.

However, scaling laws also underscore critical constraints. While larger models can achieve superior performance, the requisite resource demands increase exponentially. As illustrated in Figure 9.9, the computational demands of training state-of-the-art models are escalating at an unsustainable rate. This raises pertinent questions regarding the environmental impact and economic viability of continued scaling.

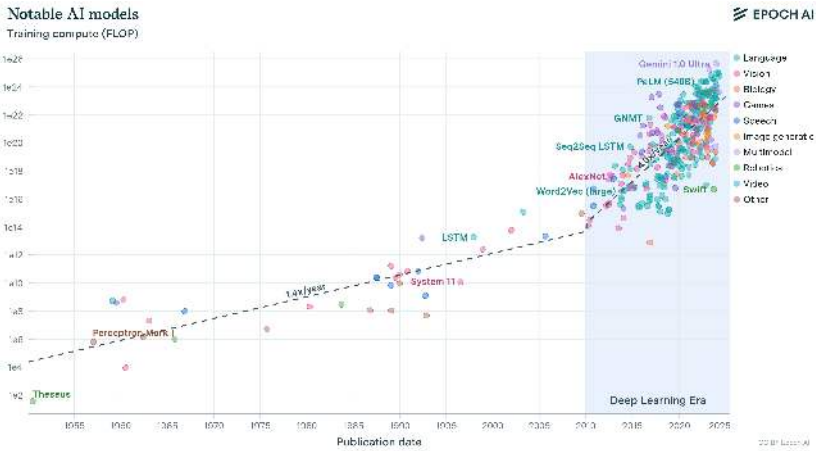


Figure 9.3: Model training compute is growing at faster and faster rates, especially in the recent deep learning era. Source: (Sevilla et al. 2022a.)

Understanding these fundamental relationships is crucial for informing critical decisions pertaining to system design and resource allocation. They delineate both the potential benefits of scaling and its inherent costs, guiding the development of more efficient and sustainable machine learning systems. This understanding provides the necessary context for our subsequent examination of algorithmic, compute, and data efficiency.

9.2.2 Empirical Scaling Laws

Scaling laws delineate the relationship between the performance of machine learning models and the augmentation of resources, including model size, dataset size, and computational budget. These relationships are typically expressed as power-law functions,[1] which demonstrate that model loss decreases predictably with increased resource allocation. Empirical investigations across diverse domains have corroborated that performance metrics, including accuracy or perplexity, exhibit smooth and monotonic improvements when models are scaled along these dimensions.

A key example of this behavior is the relationship between generalization error and dataset size, which exhibits three distinct regimes: a Small Data Region, a Power-law Region, and an Irreducible Error Region (Hestness et al. 2017). As shown in Figure 9.4, small datasets lead to high generalization error constrained

[1] Power-law behavior has been a fundamental pattern in machine learning since early neural network research. While theoretical work in the 1960s and 1980s posited capacity scaling benefits, empirical validation of these relationships across multiple orders of magnitude became feasible only with contemporary computational resources and large-scale datasets.

by poor estimates (best-guess error). As more data becomes available, models enter the power-law region, where generalization error decreases predictably as a function of dataset size. Eventually, this trend saturates, approaching the irreducible error floor, beyond which further data yields negligible improvements. This visualization demonstrates the principle of diminishing returns and highlights the operational regime in which data scaling is most effective.

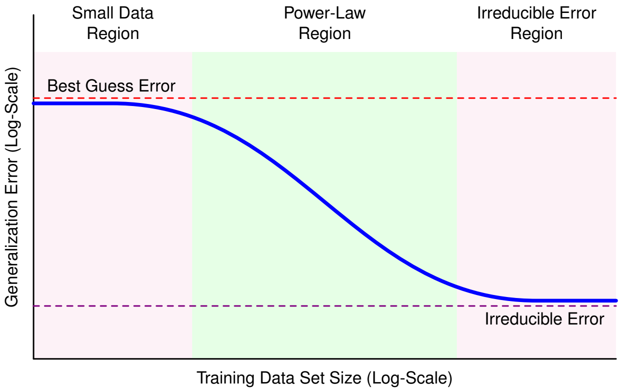


Figure 9.4: The relationship between generalization error and dataset size exhibits three distinct regimes. As dataset size increases, generalization error decreases predictably until it reaches an irreducible error floor. Source: (Hestness et al. 2017).

A general formulation of this relationship is expressed as:

$$\mathscr{L}(N) = AN^{-\alpha} + B$$

where $\mathscr{L}(N)$ represents the loss achieved with resource quantity N, A and B are task-dependent constants, and α is the scaling exponent that characterizes the rate of performance improvement. A larger value of α signifies that performance improvements are more efficient with respect to scaling. This formulation also encapsulates the principle of diminishing returns: incremental gains in performance decrease as N increases.

Empirical evidence for scaling laws is most prominently observed in large language models. In a seminal study, Kaplan et al. (2020) demonstrated that the cross-entropy loss of transformer-based language models scales predictably with three pivotal factors: the number of model parameters, the volume of the training dataset (measured in tokens), and the total computational budget (measured in floating-point operations).[2]

When these factors are augmented proportionally, models exhibit consistent performance improvements without necessitating architectural modifications or task-specific tuning. This behavior underlies contemporary training strategies for large-scale language models and has significantly influenced design decisions in both research and production environments.

These empirical patterns are illustrated in Figure 9.5, which presents test loss curves for models spanning a range of sizes, from 10^3 to 10^9 parameters. The figure reveals two key insights. First, larger models demonstrate superior sample efficiency, achieving target performance levels with fewer training tokens. Second, as computational resources increase, the optimal model size correspondingly grows, with loss decreasing predictably when compute is

[2] This study significantly altered the machine learning community's understanding of the impact of scale on model performance through comprehensive empirical validation of scaling laws. Its findings directly influenced training methodologies for large language models such as GPT-3, establishing a quantitative framework for predicting performance improvements based on compute, data, and model size.

allocated efficiently. The curves also highlight a practical consideration in large-scale training: compute-optimal solutions often entail early stopping before full convergence, demonstrating the inherent trade-off between training duration and resource utilization.

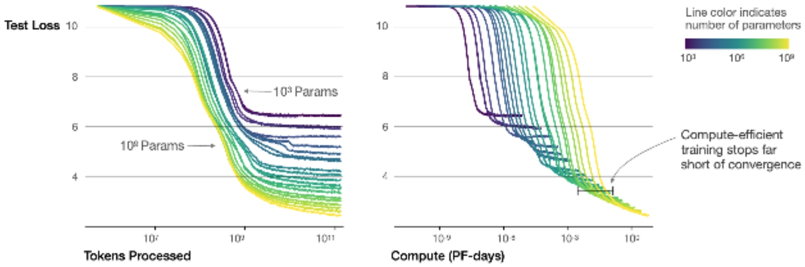


Figure 9.5: Test loss curves show that: (1) On the left, larger models achieve better performance with fewer training tokens, leading to better sample efficiency. (2) On the right, with increased compute budget, optimal model size grows steadily and loss decreases consistently. Source: Kaplan et al. (2020)

More fundamentally, this work established a theoretical scaling relationship that defines the optimal allocation of compute between model size and dataset size. For a fixed compute budget C, the optimal trade-off occurs when the dataset size D and model size N satisfy the relationship $D \propto N^{0.74}$.[3] This equation defines the compute-optimal scaling frontier, where neither the model is undertrained nor the data underutilized. Deviations from this equilibrium, for example, training a large model on insufficient data, result in suboptimal compute utilization and degraded performance. In practical terms, this implies that scaling model size alone is insufficient; proportional increases in data and compute are required to maintain efficient training dynamics.

This theoretical prediction is corroborated by empirical fits across multiple model configurations. As shown in Figure 9.6, the early-stopped test loss $\mathcal{L}(N, D)$ varies predictably with both dataset size and model size, and learning curves across configurations can be aligned through appropriate parameterization. These results further substantiate the regularity of scaling behavior and provide a practical tool for guiding model development under resource constraints.

Similar trends have been observed in other domains. In computer vision, model families such as ResNet and EfficientNet exhibit consistent accuracy improvements when scaled along dimensions of depth, width, and resolution, provided the scaling adheres to principled heuristics. These empirical patterns reinforce the observation that the benefits of scale are governed by underlying regularities that apply broadly across architectures and tasks.

As long as the scaling regime remains balanced, in which the model underfits the data while compute capacity is fully utilized, performance continues to improve predictably. However, once these assumptions are violated, scaling may lead to overfitting, underutilized resources, or inefficiencies, as explored in subsequent sections.

9.2.3 Scaling Regimes

While the scaling laws discussed thus far have focused primarily on pre-training, recent research indicates that scaling behavior extends to other phases of model

[3] The exponent 0.74 is empirically derived for transformer-based language models under autoregressive training. It represents the balance point where scaling both data and model together yields optimal performance for a fixed compute budget, and has informed practical model development pipelines across industry.

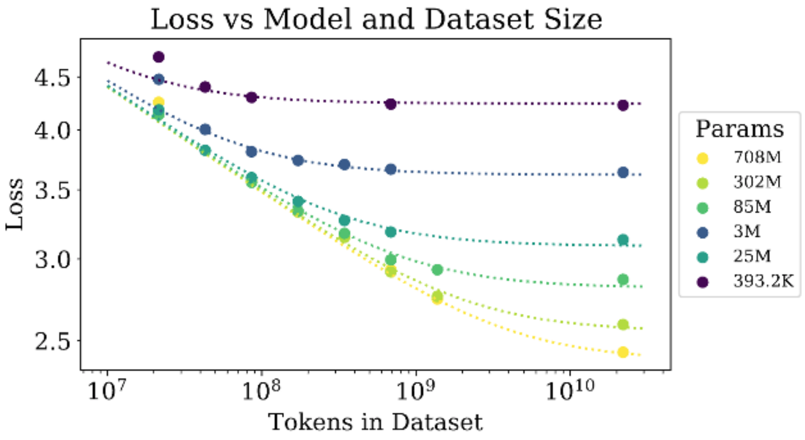


Figure 9.6: Test loss surface $\mathcal{L}(N, D)$ shows predictable variation across model size N and dataset size D.

development and deployment. A more complete understanding emerges by examining three distinct scaling regimes that characterize different stages of the machine learning pipeline.

The first regime, pre-training scaling, encompasses the traditional domain of scaling laws—how model performance improves with larger architectures, expanded datasets, and increased compute during initial training. This has been extensively studied in the context of foundation models[4], where clear power-law relationships emerge between resources and capabilities.

Post-training scaling is the second regime that focuses on improvements achieved after initial training through techniques such as fine-tuning, prompt engineering, and task-specific data augmentation. This regime has gained prominence with the rise of foundation models, where adaptation rather than retraining often provides the most efficient path to enhanced performance.

The third regime, test-time scaling, addresses how performance can be improved by allocating additional compute during inference, without modifying the model's parameters. This includes methods such as ensemble prediction, chain-of-thought prompting, and iterative refinement, which effectively allow models to spend more time processing each input.

As shown in Figure 9.7, these regimes exhibit distinct characteristics in how they trade computational resources for improved performance. Pre-training scaling typically requires massive resources but provides broad capability improvements. Post-training scaling offers more targeted enhancements with moderate resource requirements. Test-time scaling provides flexible performance-compute trade-offs that can be adjusted per inference.

Understanding these regimes is crucial for system design, as it reveals multiple paths to improving performance beyond simply scaling up model size or training data. For resource-constrained deployments, post-training and test-time scaling may provide more practical approaches than full model retraining. Similarly, in high-stakes applications, test-time scaling offers a way to trade latency for accuracy when needed.

[4] Foundation Models: Large-scale AI models pre-trained on vast amounts of data that can be adapted to a wide range of downstream tasks. Examples include GPT-3, PaLM, and BERT. These models demonstrate emergent capabilities as they scale in size and training data.

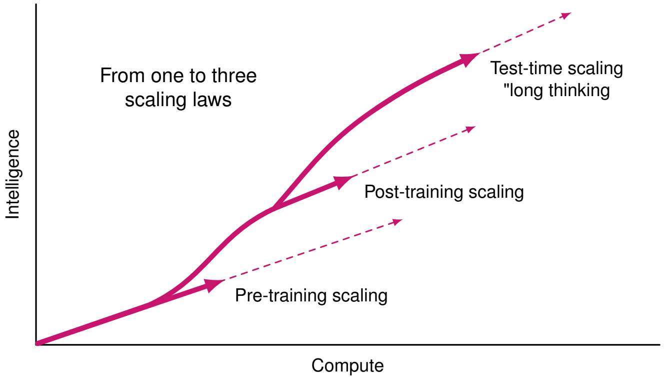


Figure 9.7: The three scaling regimes: pre-training, post-training, and test-time scaling. Each regime exhibits different compute–performance characteristics.

This framework provides a more nuanced view of scaling in machine learning systems. By considering all three regimes, designers can make more informed decisions about resource allocation and optimization strategies across the full model lifecycle. The interplay between these regimes also suggests opportunities for hybrid approaches that leverage the strengths of each scaling mode while managing their respective costs and limitations.

9.2.4 System Design

Scaling laws provide insights into the behavior of machine learning systems as resource allocation increases. The consistent observation of power-law trends suggests that, within a well-defined operational regime, model performance is predominantly determined by scale rather than idiosyncratic architectural innovations. This observation has significant ramifications for system design, resource planning, and the evaluation of efficiency.

A salient characteristic of these laws is the phenomenon of diminishing returns. While augmenting model size or training data volume yields performance improvements, the rate of these improvements diminishes with increasing scale. For instance, doubling the parameter count from 100 million to 200 million may produce substantial gains, whereas a similar doubling from 100 billion to 200 billion may yield only incremental enhancements. This behavior is mathematically captured by the scaling exponent α, which dictates the slope of the performance curve. Lower values of α indicate that more aggressive scaling is necessary to achieve comparable performance gains.

Practically, this implies that unmitigated scaling is ultimately unsustainable. Each successive increment in performance necessitates a disproportionately larger investment in data, compute, or model size. Consequently, scaling laws underscore the escalating tension between model performance and resource expenditure—a central theme of this discourse. They also emphasize the imperative of balanced scaling, wherein increments in one resource dimension (e.g., model parameters) must be accompanied by commensurate increments in other dimensions (e.g., dataset size and compute budget) to maintain optimal performance progression.

Furthermore, scaling laws can serve as a diagnostic instrument for identifying performance bottlenecks. Performance plateaus despite increased resource allocation may indicate saturation in one dimension, such as inadequate data in relation to model size, or inefficient utilization of computational resources. This diagnostic capability renders scaling laws not only predictive but also prescriptive, enabling practitioners to ascertain the optimal allocation of resources for maximum efficacy.

Understanding scaling laws is not merely of theoretical interest—it has direct implications for the practical design of efficient machine learning systems. By revealing how performance responds to increases in model size, data, and compute, scaling laws provide a principled framework for making informed design decisions across the full lifecycle of system development.

One key application is in resource budgeting. Scaling laws allow practitioners to estimate the returns on investment for different types of resources. For example, when facing a fixed computational budget, designers can use empirical scaling curves to determine whether performance gains are better achieved by increasing model size, expanding the dataset, or improving training duration. This enables more strategic allocation of limited resources, particularly in scenarios where cost, energy, or time constraints are dominant factors.

In OpenAI's development of GPT-3, the authors followed scaling laws derived from earlier experiments to determine the appropriate training dataset size and model parameter count (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, and al. 2020). Rather than conducting expensive architecture searches, they scaled a known transformer architecture along the compute-optimal frontier to 175 billion parameters and 300 billion tokens. This approach allowed them to predict model performance and resource requirements in advance, highlighting the practical value of scaling laws in large-scale system planning.

Scaling laws also inform decisions about model architecture. Rather than relying on exhaustive architecture search or ad hoc heuristics, system designers can use scaling trends to identify when architectural changes are likely to yield significant improvements and when gains are better pursued through scale alone. For instance, if a given model family follows a favorable scaling curve, it may be preferable to scale that architecture rather than switching to a more complex but untested design. Conversely, if scaling saturates early, it may indicate that architectural innovations are needed to overcome current limitations.

Moreover, scaling laws can guide deployment strategy. In edge and embedded environments, system designers often face tight resource budgets. By understanding how performance degrades when a model is scaled down, it is possible to choose smaller configurations that deliver acceptable accuracy within the deployment constraints. This supports the use of model families with predictable scaling properties, enabling a continuum of options from high-performance cloud deployment to lightweight on-device inference.

Finally, scaling laws provide insight into efficiency limits. By quantifying the trade-offs between scale and performance, they highlight when brute-force scaling becomes inefficient and signal the need for alternative approaches. This includes methods such as knowledge distillation, transfer learning, sparsity, and hardware-aware model design—all of which aim to extract more value from existing resources without requiring further increases in raw scale.

In this way, scaling laws serve as a compass for system designers, helping them navigate the complex landscape of performance, efficiency, and practicality. They do not dictate a single path forward, but they provide the analytical foundation for choosing among competing options in a principled and data-driven manner.

9.2.5 Scaling vs. Efficiency

While scaling laws elucidate a pathway to performance enhancement through the augmentation of model size, dataset volume, and computational budget, they concurrently reveal the rapidly escalating resource demands associated with such progress. As models become increasingly large and sophisticated, the resources necessary to support their training and deployment expand disproportionately. This phenomenon introduces a fundamental tension within contemporary machine learning: the performance gains achieved through scaling are often accompanied by a significant compromise in system efficiency.

A primary concern is the computational expenditure. Training large-scale models necessitates substantial processing power, typically requiring distributed infrastructures comprising hundreds or thousands of accelerators. For instance, the training of state-of-the-art language models may require tens of thousands of GPU-days, consuming millions of kilowatt-hours of electricity and incurring financial costs that are prohibitive for many institutions. As previously discussed, the energy demands of training have outpaced Moore's Law, raising critical questions regarding the long-term sustainability of continued scaling.

In addition to computational resources, data acquisition and curation present significant trade-offs. Large models demand not only extensive data volumes but also high-quality, diverse datasets to realize their full potential. The collection, cleansing, and labeling of such datasets are both time-consuming and costly. Furthermore, as models approach saturation of available high-quality data, especially in fields like natural language processing, further performance gains through data scaling become increasingly challenging. This necessitates a focus on extracting greater value from existing data, emphasizing the importance of data efficiency as a complement to brute-force scaling.

The financial and environmental implications of scaling also warrant careful consideration. Training runs for large foundation models can incur millions of U.S. dollars in computational expenses alone, and the carbon footprint associated with such training has garnered increasing scrutiny. These costs limit accessibility to cutting-edge research and exacerbate disparities in access to advanced AI systems. From a system design perspective, this underscores the imperative to develop more resource-efficient scaling strategies that minimize consumption without sacrificing performance.

Collectively, these trade-offs highlight that while scaling laws provide a valuable framework for understanding performance growth, they do not offer an unencumbered path to improvement. Each incremental performance gain must be evaluated against the corresponding resource requirements. As machine learning systems approach the practical limits of scale, the focus must shift from mere scaling to efficient scaling. This transition necessitates a holistic approach to system design that balances performance, cost, energy, and environmental impact, ensuring that advancements in AI are not only effective but also sustainable and equitable.

9.2.6 Scaling Breakdown

While scaling laws exhibit remarkable consistency within specific operational regimes, they are not devoid of limitations. As machine learning systems expand, they inevitably encounter boundaries where the underlying assumptions of smooth, predictable scaling no longer hold. These breakdown points reveal critical inefficiencies and underscore the necessity for more refined system design.

A common failure mode is imbalanced scaling. For scaling laws to remain valid, model size, dataset size, and computational budget must be augmented in a coordinated manner. Over-investment in one dimension while maintaining others constant often results in suboptimal outcomes. For example, increasing model size without expanding the training dataset may induce overfitting, whereas increasing computational resources without model redesign may lead to inefficient resource utilization (Hoffmann et al. 2022). In such scenarios, performance plateaus or even declines despite increased resource expenditure.

Closely related is the issue of underutilized compute budgets. Large-scale models often require carefully tuned training schedules and learning rates to make full use of available resources. When compute is insufficiently allocated, due to premature stopping, batch size misalignment, or ineffective parallelism, models may fail to reach their performance potential despite significant infrastructure investment.

Another prevalent mode of failure is data saturation. Scaling laws presuppose that model performance will continue to improve with access to sufficient training data. However, in numerous domains, particularly in the fields of language and vision, the availability of high-quality, human-annotated data is finite. As models consume increasingly large datasets, they eventually reach a point of diminishing marginal utility, where additional data points contribute minimal new information. Beyond this threshold, larger models may exhibit memorization rather than generalization, leading to degraded performance on out-of-distribution tasks. This issue is particularly acute when scaling is pursued without commensurate enhancements in data diversity or quality.

Infrastructure bottlenecks also impose practical scaling constraints. As models grow in size, they demand greater memory bandwidth, interconnect capacity, and I/O throughput. These hardware limitations become increasingly challenging to overcome, even with specialized accelerators. For instance, distributing a trillion-parameter model across a cluster necessitates meticulous management of data parallelism, communication overhead, and fault tolerance. The complexity of orchestrating such large-scale systems introduces engineering challenges that can diminish the theoretical gains predicted by scaling laws.

Finally, semantic saturation presents a significant conceptual challenge. At extreme scales, models may approach the limits of what can be learned from their training distributions. Performance on benchmark tasks may continue to improve, but these improvements may no longer reflect meaningful gains in generalization or understanding. Instead, models may become increasingly brittle, susceptible to adversarial examples, or prone to generating plausible but inaccurate outputs—particularly in generative tasks.

These breakdown points demonstrate that scaling laws, while powerful, are not absolute. They describe empirical regularities under specific conditions, which become increasingly difficult to maintain at scale. As machine learning systems continue to evolve, it is essential to discern where and why scaling ceases to be effective—and to develop strategies that enhance performance without relying solely on scale.

To synthesize the primary causes of scaling failure, the following diagnostic matrix (Table 9.1) outlines typical breakdown types, their underlying causes, and representative scenarios. This table serves as a reference point for anticipating inefficiencies and guiding more balanced system design.

Table 9.1: Common failure modes associated with unbalanced or excessive scaling across model, data, and compute dimensions.

Dimension Scaled	Type of Breakdown	Underlying Cause	Example Scenario
Model Size	Overfitting	Model capacity exceeds available data	Billion-parameter model on limited dataset
Data Volume	Diminishing Returns	Saturation of new or diverse information	Scaling web text beyond useful threshold
Compute Budget	Underutilized Resources	Insufficient training steps or inefficient use	Large model with truncated training duration
Imbalanced Scaling	Inefficiency	Uncoordinated increase in model/data/compute	Doubling model size without more data or time
All Dimensions	Semantic Saturation	Exhaustion of learnable patterns in the domain	No further gains despite scaling all inputs

In this section, we have explored the fundamental principles of AI scaling laws, examining their empirical foundations, practical implications, and inherent limitations. Scaling laws provide a valuable framework for understanding how model performance scales with resources, but they also highlight the importance of efficiency and sustainability. The trade-offs and challenges associated with scaling underscore the need for a holistic approach to system design, which balances performance with resource constraints. In the following sections, we will delve into the specific dimensions of efficiency, including algorithmic, computational, and data-related aspects, exploring how these areas contribute to the development of more sustainable and effective machine learning systems.

9.2.7 Toward Efficient Scaling

While the empirical success of scaling laws has driven substantial progress in artificial intelligence, these observations raise foundational questions that extend beyond resource allocation. The sustainability of the current scaling trajectory, and its adequacy in capturing the principles of efficient AI system design, must be critically examined.

The empirical regularities observed in scaling laws prompt deeper inquiry: can observation be translated into a theoretical framework that explains the mechanisms driving these patterns? Establishing such a framework would enhance scientific understanding and inform the design of more efficient algorithms and architectures. The limitations inherent in brute-force scaling, including diminishing returns and observed breakdowns, highlight the need for architectural innovations that reshape scaling behavior and address efficiency from an algorithmic standpoint.

Simultaneously, the increasing demand for data emphasizes the importance of transitioning to a data-centric paradigm. As data saturation is approached, the efficiency of data acquisition, curation, and utilization becomes critical. This shift requires a deeper understanding of data dynamics and the development of strategies that maximize the utility of limited data resources. These considerations form the basis for the upcoming discussion on data efficiency.

The computational demands of large-scale models further underscore the necessity of compute efficiency. Sustainable scaling depends on minimizing resource consumption while maintaining or improving performance. This objective motivates the exploration of hardware-optimized architectures and training methodologies that support efficient execution, to be discussed in the context of compute efficiency.

Algorithmic, compute, and data efficiency are not independent; their interdependence shapes the overall performance of machine learning systems. The emergence of novel capabilities in extremely large models suggests the potential for synergistic effects across these dimensions. Achieving real-world efficiency requires a holistic approach to system design in which these elements are carefully orchestrated. This perspective introduces the forthcoming discussion on system efficiency.

Finally, the ethical considerations surrounding access to compute and data resources demonstrate that efficiency is not solely a technical goal. Ensuring equitable distribution of the benefits of efficient AI represents a broader societal imperative. Subsequent sections will address these challenges, including the limits of optimization, the implications of Moore's Law, and the balance between innovation and accessibility.

In summary, the future of scaling lies not in unbounded expansion but in the coordinated optimization of algorithmic, compute, and data resources. The sections that follow will examine each of these dimensions and their contributions to the development of efficient and sustainable machine learning systems. Although scaling laws offer a valuable perspective, they represent only one component of a more comprehensive framework.

9.3 The Pillars of AI Efficiency

The trajectory of machine learning has been significantly shaped by scaling laws and the evolving concept of efficiency. While scaling laws demonstrate the potential benefits of increasing model size, dataset volume, and computational resources, they also highlight the critical need for efficient resource utilization. To systematically address these challenges, we delineate three fundamental and interconnected pillars of AI efficiency: algorithmic efficiency, compute efficiency, and data efficiency. These pillars represent critical domains that have profoundly influenced how we navigate the trade-offs revealed by scaling laws.

Algorithmic efficiency pertains to the design and optimization of algorithms to maximize performance within given resource constraints. As scaling laws indicate that larger models generally perform better, algorithmic efficiency becomes crucial for making these models practical and deployable. Contemporary research focuses on techniques such as model compression, architectural optimization, and algorithmic refinement, all aimed at preserving the benefits of scale while minimizing resource consumption.

Compute efficiency addresses the optimization of computational resources, including hardware and energy utilization. Scaling laws have shown that training compute requirements are growing at an exponential rate, making compute efficiency increasingly critical. The advent of specialized hardware accelerators, such as GPUs and TPUs, has enabled the development of large-scale models. However, the energy demands associated with training and deploying these models have raised concerns regarding sustainability. Compute efficiency, therefore, encompasses strategies for optimizing hardware utilization, reducing energy footprint, and exploring alternative computing paradigms that can support continued scaling.

Data efficiency focuses on maximizing the information gained from available data while minimizing the required data volume. Scaling laws demonstrate that model performance improves with larger datasets, but they also reveal diminishing returns and practical limits to data collection. This pillar becomes especially important as we approach the boundaries of available high-quality data in domains like language modeling. Methods such as data augmentation, active learning, and efficient data representation aim to achieve the benefits predicted by scaling laws with reduced data requirements.

These three pillars are not mutually exclusive; rather, they are deeply intertwined and often mutually reinforcing. Improvements in one pillar can lead to gains in others, and trade-offs between them are frequently necessary. As we examine the historical evolution of these dimensions, as depicted in Figure 9.8, we will elucidate the dynamic interplay between algorithmic, compute, and data efficiency, providing a foundation for understanding how to achieve efficient scaling in contemporary machine learning.

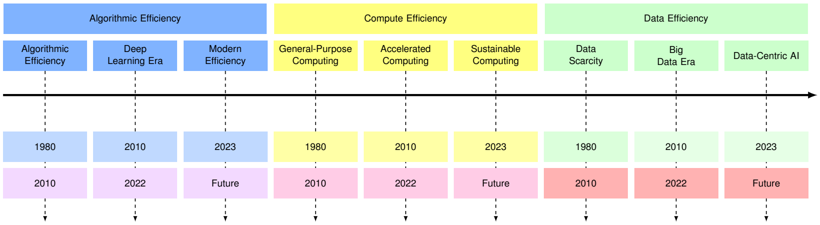


Figure 9.8: Evolution of AI Efficiency over the past few decades.

9.3.1 Algorithmic Efficiency

Model efficiency addresses the design and optimization of machine learning models to deliver high performance while minimizing computational and memory requirements. It is a critical component of machine learning systems, enabling models to operate effectively across a range of platforms, from cloud servers to resource-constrained edge devices. The evolution of algorithmic efficiency mirrors the broader trajectory of machine learning itself, shaped by algorithmic advances, hardware developments, and the increasing complexity of real-world applications.

9.3.1.1 Early Efficiency

During the early decades of machine learning, algorithmic efficiency was closely tied to computational constraints, particularly in terms of parallelization. Early algorithms like decision trees and SVMs were primarily optimized for single-machine performance, with parallel implementations limited mainly to ensemble methods where multiple models could be trained independently on different data batches.

Neural networks also began to emerge during this period, but they were constrained by the limited computational capacity of the time. Unlike earlier algorithms, neural networks showed potential for model parallelism, the ability to distribute model components across multiple processors, though this advantage wouldn't be fully realized until the deep learning era. This led to careful optimizations in their design, such as limiting the number of layers or neurons to keep computations manageable. Efficiency was achieved not only through model simplicity but also through innovations in optimization techniques, such as the adoption of stochastic gradient descent, which made training more practical for the hardware available.

The era of algorithmic efficiency laid the groundwork for machine learning by emphasizing the importance of achieving high performance under strict resource constraints. These principles remain important even in today's datacenter-scale computing, where hardware limitations in memory bandwidth and power consumption continue to drive innovation in algorithmic efficiency. It was an era of problem-solving through mathematical rigor and computational restraint, establishing patterns that would prove valuable as models grew in scale and complexity.

9.3.1.2 Deep Learning Era

The introduction of deep learning in the early 2010s marked a turning point for algorithmic efficiency. Neural networks, which had previously been constrained by hardware limitations, now benefited from advancements in computational power, particularly the adoption of GPUs (Krizhevsky, Sutskever, and Hinton 2017b). This capability allowed researchers to train larger, more complex models, leading to breakthroughs in tasks such as image recognition, natural language processing, and speech synthesis.

However, the growing size and complexity of these models introduced new challenges. Larger models required significant computational resources and memory, making them difficult to deploy in practical applications. To address these challenges, researchers developed techniques to reduce model size and computational requirements without sacrificing accuracy. Pruning[5], for instance, involved removing redundant or less significant connections within a

[5] Pruning was inspired by biological development where unused connections between neurons are eliminated during brain development.

neural network, reducing both the model's parameters and its computational overhead (Yann LeCun, Denker, and Solla 1989). Quantization focused on lowering the precision of numerical representations, enabling models to run faster and with less memory (Jacob et al. 2018a). Knowledge distillation allowed large, resource-intensive models (referred to as "teachers") to transfer their knowledge to smaller, more efficient models (referred to as "students"), achieving comparable performance with reduced complexity (Hinton, Vinyals, and Dean 2015a).

At the same time, new architectures specifically designed for efficiency began to emerge. Models such as MobileNet (A. G. Howard et al. 2017a), EfficientNet (Tan and Le 2019b), and SqueezeNet (Iandola et al. 2016) demonstrated that compact designs could deliver high performance, enabling their deployment on devices with limited computational power, such as smartphones and IoT devices[6].

9.3.1.3 Modern Efficiency

As machine learning systems continue to grow in scale and complexity, the focus on algorithmic efficiency has expanded to address sustainability and scalability. Today's challenges require balancing performance with resource efficiency, particularly as models like GPT-4 and beyond are applied to increasingly diverse tasks and environments. One emerging approach involves sparsity, where only the most critical parameters of a model are retained, significantly reducing computational and memory demands. Hardware-aware design has also become a priority, as researchers optimize models to take full advantage of specific accelerators, such as GPUs, TPUs, and edge processors. Another important trend is parameter-efficient fine-tuning, where large pre-trained models can be adapted to new tasks by updating only a small subset of parameters. Low-Rank Adaptation (LoRA)[7] and prompt-tuning exemplify this approach, allowing systems to achieve task-specific performance while maintaining the efficiency advantages of smaller models.

As shown in Figure 9.9, model training compute requirements have been growing at an accelerating rate, especially in the deep learning era. This trend underscores the necessity for algorithmic innovations that enhance efficiency without compromising performance.

These advancements reflect a broader shift in focus: from scaling models indiscriminately to creating architectures that are purpose-built for efficiency. This modern era emphasizes not only technical excellence but also the practicality and sustainability of machine learning systems.

9.3.1.4 Efficiency in Design

Model efficiency is fundamental to the design of scalable and sustainable machine learning systems. By reducing computational and memory demands, efficient models lower energy consumption and operational costs, making machine learning systems accessible to a wider range of applications and deployment environments. Moreover, algorithmic efficiency complements other dimensions of efficiency, such as compute and data efficiency, by reducing the overall burden on hardware and enabling faster training and inference cycles.

[6] MobileNet/EfficientNet/SqueezeNet: Compact neural network architectures designed for efficiency, balancing high performance with reduced computational demands. MobileNet introduced depthwise separable convolutions (2017), EfficientNet applied compound scaling (2019), and SqueezeNet focused on reducing parameters using 1x1 convolutions (2016).

[7] Low-Rank Adaptation (LoRA): A technique that adapts large pre-trained models to new tasks by updating only a small subset of parameters, significantly reducing computational and memory requirements.

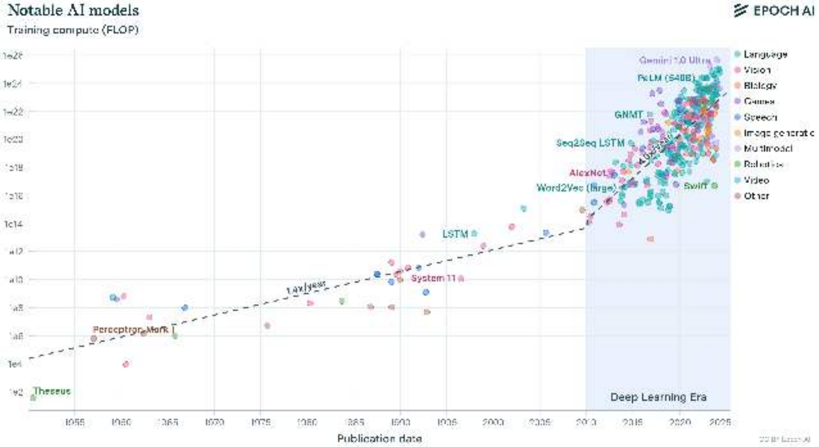


Figure 9.9: Model training compute is growing at faster and faster rates, especially in the recent deep learning era. Source: (Sevilla et al. 2022a.)

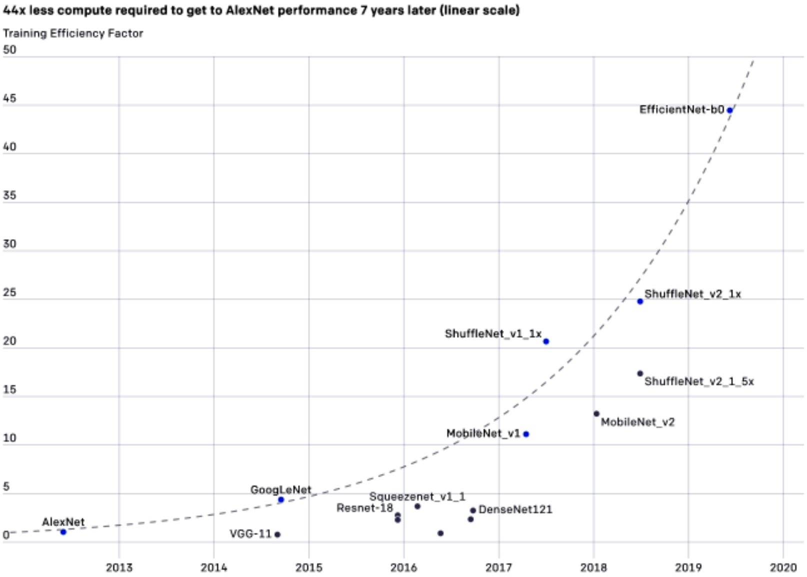


Figure 9.10: Within just seven years, 44 times less compute was required to achieve AlexNet performance. Source: (Jaech et al. 2024).

Notably, as Figure 9.10 shows, the computational resources needed to train a neural network to achieve AlexNet-level performance on ImageNet classification had decreased by $44\times$ compared to 2012. This improvement, which halved every 16 months, outpaced the hardware efficiency gains of Moore's Law[8]. Such rapid progress demonstrates the role of algorithmic advancements in driving efficiency alongside hardware innovations (Hernandez, Brown, et al. 2020).

The evolution of algorithmic efficiency, from algorithmic innovations to hardware-aware optimization, is of importance in machine learning. As the field advances, algorithmic efficiency will remain central to the design of systems that are high-performing, scalable, and sustainable.

9.3.2 Compute Efficiency

Compute efficiency focuses on the effective use of hardware and computational resources to train and deploy machine learning models. It encompasses strategies for reducing energy consumption, optimizing processing speed, and leveraging hardware capabilities to achieve scalable and sustainable system performance. The evolution of compute efficiency is closely tied to advancements in hardware technologies, reflecting the growing demands of machine learning applications over time.

9.3.2.1 General-Purpose Computing Era

In the early days of machine learning, compute efficiency was shaped by the limitations of general-purpose CPUs. During this period, machine learning models had to operate within strict computational constraints, as specialized hardware for machine learning did not yet exist. Efficiency was achieved through algorithmic innovations, such as simplifying mathematical operations, reducing model size, and optimizing data handling to minimize computational overhead.

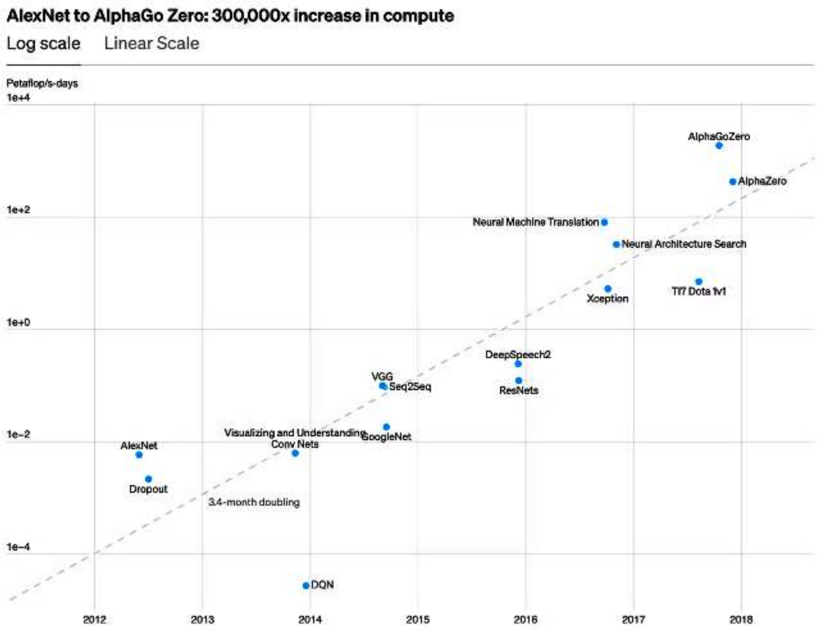
Researchers worked to maximize the capabilities of CPUs by using parallelism where possible, though options were limited. Training times for models were often measured in days or weeks, as even relatively small datasets and models pushed the boundaries of available hardware. The focus on compute efficiency during this era was less about hardware optimization and more about designing algorithms that could run effectively within these constraints.

9.3.2.2 Accelerated Computing Era

The introduction of deep learning in the early 2010s brought a seismic shift in the landscape of compute efficiency. Models like AlexNet and ResNet showed the potential of neural networks, but their computational demands quickly surpassed the capabilities of traditional CPUs. As shown in Figure 9.11, this marked the beginning of an era of exponential growth in compute usage. OpenAI's analysis reveals that the amount of compute used in AI training has increased 300,000 times since 2012, doubling approximately every 3.4 months—a rate far exceeding Moore's Law (Amodei, Hernandez, et al. 2018).

This rapid growth was driven not only by the adoption of GPUs, which offered unparalleled parallel processing capabilities, but also by the willingness

[8] Moore's Law: An observation made by Gordon Moore in 1965, stating that the number of transistors on a microchip doubles approximately every two years, leading to an exponential increase in computational power and a corresponding decrease in relative cost.



The total amount of compute, in petaflop/s-days,[D] used to train selected results that are relatively well known, used a lot of compute for their time, and gave enough information to estimate the compute used.

Figure 9.11: From AlexNet to AlphaGo Zero, there has been a 300,000x increase in demand for computing power over seven years. Source: (Yann LeCun, Bengio, and Hinton 2015b).

of researchers to scale up experiments by using large GPU clusters. Specialized hardware accelerators such as Google's Tensor Processing Units (TPUs)[9] and application-specific integrated circuits (ASICs)[10] further revolutionized compute efficiency. These innovations enabled significant reductions in training times for deep learning models, transforming tasks that once took weeks into operations completed in hours or days.

The rise of large-scale compute also highlighted the complementary relationship between algorithmic innovation and hardware efficiency. Advances such as neural architecture search and massive batch processing leveraged the increasing availability of computational power, demonstrating that more compute could directly lead to better performance in many domains.

9.3.2.3 Sustainable Computing Era

As machine learning systems scale further, compute efficiency has become closely tied to sustainability. Training state-of-the-art models like GPT-4 requires massive computational resources, leading to increased attention on the environmental impact of large-scale computing. The projected electricity usage of data centers, shown in Figure 9.12, highlights this concern. Between 2010 and 2030, electricity consumption is expected to rise sharply, particularly under the "Worst" scenario, where it could exceed 8,000 TWh by 2030[11].

The dramatic demand for energy usage underscores the urgency for compute efficiency, as even large data centers face energy constraints due to limitations in electrical grid capacity and power availability in specific locations. To address

[9] Tensor Processing Units (TPUs): Google's custom-designed AI accelerator chips, introduced in 2016, demonstrated significant performance gains—processing AI workloads up to 30 times faster than contemporary GPUs and 80 times faster than CPUs.

[10] Application-Specific Integrated Circuits (ASICs): Custom-designed chips optimized for specific machine learning workloads, offering superior performance and energy efficiency compared to general-purpose processors. ASICs can achieve 10-100x better performance per watt than GPUs for targeted applications.

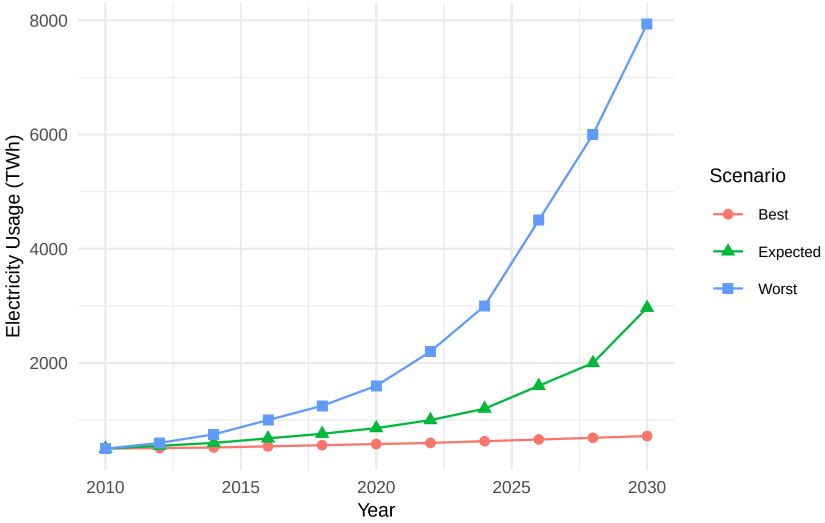


Figure 9.12: Electricity usage (TWh) of Data Centers from 2010 to 2030. Source: Andrae and Edler (2015).

these challenges, the focus today is on optimizing hardware utilization and minimizing energy consumption, both in cloud data centers and at the edge.

One key trend is the adoption of energy-aware scheduling and resource allocation techniques, which ensure that computational workloads are distributed efficiently across available hardware (D. Patterson et al. 2021a). Researchers are also developing methods to dynamically adjust precision levels during training and inference, using lower precision operations (e.g., mixed-precision training) to reduce power consumption without sacrificing accuracy.

Another focus is on distributed systems, where compute efficiency is achieved by splitting workloads across multiple machines. Techniques such as model parallelism and data parallelism allow large-scale models to be trained more efficiently, leveraging clusters of GPUs or TPUs to maximize throughput. These methods reduce training times while minimizing the idle time of hardware resources.

At the edge, compute efficiency is evolving to address the growing demand for real-time processing in energy-constrained environments. Innovations such as hardware-aware model optimization, lightweight inference engines, and adaptive computing architectures are paving the way for highly efficient edge systems. These advancements are critical for enabling applications like autonomous vehicles and smart home devices, where latency and energy efficiency are paramount.

9.3.2.4 Compute Efficiency's Role

Compute efficiency is a critical enabler of system-wide performance and scalability. By optimizing hardware utilization and energy consumption, it ensures that machine learning systems remain practical and cost-effective, even as models and datasets grow larger. Moreover, compute efficiency directly complements

[11] The “Best,” “Expected,” and “Worst” scenarios in the figure reflect different assumptions about how efficiently data centers can handle increasing internet traffic, with the best-case scenario assuming the fastest improvements in energy efficiency and the worst-case scenario assuming minimal gains, leading to sharply rising energy demands.

model and data efficiency. For example, compact models reduce computational requirements, while efficient data pipelines streamline hardware usage.

The evolution of compute efficiency highlights its essential role in addressing the growing demands of modern machine learning systems. From early reliance on CPUs to the emergence of specialized accelerators and sustainable computing practices, this dimension remains central to building scalable, accessible, and environmentally responsible machine learning systems.

9.3.3 Data Efficiency

Data efficiency focuses on optimizing the amount and quality of data required to train machine learning models effectively. As datasets have grown in scale and complexity, managing data efficiently has become an increasingly critical challenge for machine learning systems. While historically less emphasized than model or compute efficiency, data efficiency has emerged as a pivotal dimension, driven by the rising costs of data collection, storage, and processing. Its evolution reflects the changing role of data in machine learning, from a scarce resource to a massive but unwieldy asset.

9.3.3.1 Data Scarcity Era

In the early days of machine learning, data efficiency was not a significant focus, largely because datasets were relatively small and manageable. The challenge during this period was often acquiring enough labeled data to train models effectively. Researchers relied heavily on curated datasets, such as UCI's Machine Learning Repository, which provided clean, well-structured data for experimentation. Feature selection and dimensionality reduction techniques, such as principal component analysis (PCA), were common methods for ensuring that models extracted the most valuable information from limited data.

During this era, data efficiency was achieved through careful preprocessing and data cleaning. Algorithms were designed to work well with relatively small datasets such as MNIST (L. Deng 2012), Caltech 101 (Fei-Fei, Fergus, and Perona, n.d.) and CIFAR-10 (Krizhevsky 2009), and computational limitations reinforced the need for data parsimony. These constraints shaped the development of techniques that maximized performance with minimal data, ensuring that every data point contributed meaningfully to the learning process.

9.3.3.2 Big Data Era

The advent of deep learning in the 2010s transformed the role of data in machine learning. Models such as AlexNet and GPT-3 demonstrated that larger datasets often led to better performance, particularly for complex tasks like image classification and natural language processing. This marked the beginning of the "big data" era, where the focus shifted from making the most of limited data to scaling data collection and processing to unprecedented levels.

However, this reliance on large datasets introduced significant inefficiencies. Data collection became a costly and time-consuming endeavor, requiring vast amounts of labeled data for supervised learning tasks. To address these challenges, researchers developed techniques to enhance data efficiency, even as datasets continued to grow. Transfer learning allowed pre-trained models to be fine-tuned on smaller datasets, reducing the need for task-specific data (Yosinski et al. 2014). Data augmentation techniques, such as image rotations or text paraphrasing, artificially expanded datasets by creating new variations of existing samples. Additionally, active learning[12] prioritized labeling only the most informative data points, minimizing the overall labeling effort while maintaining performance (Settles 2012a).

Despite these advancements, the "more data is better" paradigm dominated this period, with less attention paid to streamlining data usage. As a result, the environmental and economic costs of managing large datasets began to emerge as significant concerns.

9.3.3.3 Modern Data Efficiency Era

As machine learning systems grow in scale, the inefficiencies of large datasets have become increasingly apparent. Recent work has focused on developing approaches that maximize the value of data while minimizing resource requirements. This shift reflects a growing understanding that bigger datasets do not always lead to better performance, particularly when considering the computational and environmental costs of training on massive scales.

Data-centric AI has emerged as a key paradigm, emphasizing the importance of data quality over quantity. This approach focuses on enhancing data preprocessing, removing redundancy, and improving labeling efficiency. Research has shown that careful curation and filtering of datasets can achieve comparable or superior model performance while using only a fraction of the original data volume. For instance, systematic analyses of web-scale datasets demonstrate that targeted filtering techniques can maintain model capabilities while significantly reducing training data requirements (Penedo et al. 2024).

Several techniques have emerged to support this transition toward data efficiency. Self-supervised learning enables models to learn meaningful representations from unlabeled data, reducing the dependency on expensive human-labeled datasets. Active learning strategies selectively identify the most informative examples for labeling, while curriculum learning structures the training process to progress from simple to complex examples, improving learning efficiency. These approaches work together to minimize data requirements while maintaining model performance.

The importance of data efficiency is particularly evident in foundation models. As these models grow in scale and capability, they are approaching the limits of available high-quality training data, especially for language tasks (Figure 9.13). This scarcity drives innovation in data processing and curation techniques, pushing the field to develop more sophisticated approaches to data efficiency.

Evidence for the impact of data quality appears across different scales of deployment. In Tiny ML applications, datasets like Wake Vision demonstrate how model performance critically depends on careful data curation (C. Banbury et al. 2024). At larger scales, research on language models trained on web-scale datasets shows that intelligent filtering and selection strategies can significantly improve performance on downstream tasks (Penedo et al. 2024).

This modern era of data efficiency represents a fundamental shift in how machine learning systems approach data utilization. By focusing on quality over

[12] A machine learning approach where the model selects the most informative data points for labeling to improve learning efficiency.

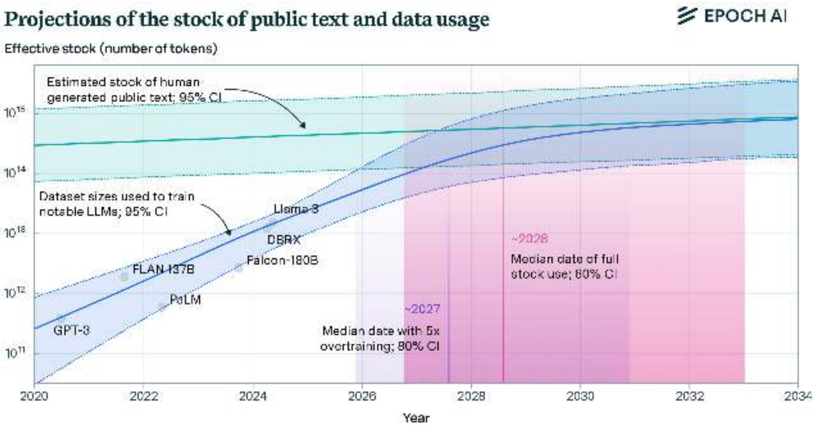


Figure 9.13: Datasets for foundation model training are quickly growing in size and capturing most of stock of human-generated text. Source: Sevilla et al. (2022b).

quantity and developing sophisticated techniques for data selection and processing, the field is moving toward more sustainable and effective approaches to model training and deployment.

9.3.3.4 Data Efficiency's Role

Data efficiency is integral to the design of scalable and sustainable machine learning systems. By reducing the dependency on large datasets, data efficiency directly impacts both model and compute efficiency. For instance, smaller, higher-quality datasets reduce training times and computational demands, while enabling models to generalize more effectively. This dimension of efficiency is particularly critical for edge applications, where bandwidth and storage limitations make it impractical to rely on large datasets.

As the field advances, data efficiency will play an increasingly prominent role in addressing the challenges of scalability, accessibility, and sustainability. By rethinking how data is collected, processed, and utilized, machine learning systems can achieve higher levels of efficiency across the entire pipeline.

9.4 System Efficiency

The efficiency of machine learning systems has become a crucial area of focus. Optimizing these systems helps us ensure that they are not only high-performing but also adaptable, cost-effective, and environmentally sustainable. Understanding the concept of ML system efficiency, its key dimensions, and the interplay between them is essential for uncovering how these principles can drive impactful, scalable, and responsible AI solutions.

9.4.1 Defining System Efficiency

Machine learning is a highly complex field, involving a multitude of components across a vast domain. Despite its complexity, there has not been a synthesis of what it truly means to have an efficient machine learning system. Here, we take a first step towards defining this concept.

ℹ Definition of Machine Learning System Efficiency

Machine Learning System Efficiency refers to the optimization of machine learning systems across three interconnected dimensions—*algorithmic efficiency*, *compute efficiency*, and *data efficiency*. Its goal is to minimize *computational, memory, and energy* demands while maintaining or improving system performance. This efficiency ensures that machine learning systems are *scalable, cost-effective, and sustainable*, which allows them to adapt to diverse deployment contexts, ranging from *cloud data centers* to *edge devices*. Achieving system efficiency, however, often requires navigating *trade-offs* between dimensions, such as balancing *model complexity* with *hardware constraints* or reducing *data dependency* without compromising *generalization*.

This definition highlights the holistic nature of efficiency in machine learning systems, emphasizing that the three dimensions, algorithmic efficiency, compute efficiency, and data efficiency, are deeply interconnected. Optimizing one dimension often affects the others, either by creating synergies or necessitating trade-offs. Understanding these interdependencies is essential for designing systems that are not only performant but also scalable, adaptable, and sustainable (D. Patterson et al. 2021b).

To better understand this interplay, we must examine how these dimensions reinforce one another and the challenges in balancing them. While each dimension contributes uniquely, the true complexity lies in their interdependencies. Historically, optimizations were often approached in isolation. However, recent years have seen a shift towards co-design, where multiple dimensions are optimized concurrently to achieve superior overall efficiency.

9.4.2 Efficiency Interdependencies

The efficiency of machine learning systems is inherently a multifaceted challenge that encompasses model design, computational resources, and data utilization. These dimensions, including algorithmic efficiency, compute efficiency, and data efficiency, are deeply interdependent, forming a dynamic ecosystem where improvements in one area often ripple across the others. Understanding these interdependencies is crucial for building scalable, cost-effective, and high-performing systems that can adapt to diverse application demands.

This interplay is best captured through a conceptual visualization. Figure 9.14 illustrates how these efficiency dimensions overlap and interact with each other in a simple Venn diagram. Each circle represents one of the efficiency dimensions, and their intersections highlight the areas where they influence one another, which we will explore next.

9.4.2.1 Algorithmic Efficiency Aids Compute and Data

Model efficiency is essential for efficient machine learning systems. By designing compact and streamlined models, we can significantly reduce computational demands, leading to faster and more cost-effective inference. These

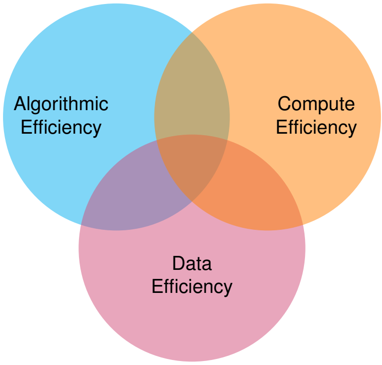


Figure 9.14: Interdependence of the different efficiency dimensions.

compact models not only consume fewer resources but are also easier to deploy across diverse environments, such as resource-constrained edge devices or energy-intensive cloud infrastructure.

Moreover, efficient models often require less data for training, as they avoid over-parameterization and focus on capturing essential patterns within the data. This results in shorter training times and reduced dependency on massive datasets, which can be expensive and time-consuming to curate. As a result, optimizing algorithmic efficiency creates a ripple effect, enhancing both compute and data efficiency.

Mobile Deployment Example. Mobile devices, such as smartphones, provide an accessible introduction to the interplay of efficiency dimensions. Consider a photo-editing application that uses machine learning to apply real-time filters. Compute efficiency is achieved through hardware accelerators like mobile GPUs or Neural Processing Units (NPUs), ensuring tasks are performed quickly while minimizing battery usage.

This compute efficiency, in turn, is supported by algorithmic efficiency. The application relies on a lightweight neural network architecture, such as MobileNets, that reduces the computational load, allowing it to take full advantage of the mobile device's hardware. Streamlined models also help reduce memory consumption, further enhancing computational performance and enabling real-time responsiveness.

Furthermore, data efficiency strengthens both compute and algorithmic efficiency by ensuring the model is trained on carefully curated and augmented datasets. These datasets allow the model to generalize effectively, reducing the need for extensive retraining and lowering the demand for computational resources during training. Additionally, by minimizing the complexity of the training data, the model can remain lightweight without sacrificing accuracy, reinforcing both model and compute efficiency.

Integrating these dimensions means mobile deployments achieve a seamless balance between performance, energy efficiency, and practicality. The interdependence of model, compute, and data efficiencies ensures that even resource-constrained devices can deliver advanced AI capabilities to users on the go.

9.4.2.2 Compute Efficiency Supports Model and Data

Compute efficiency is a key factor in optimizing machine learning systems. By maximizing hardware utilization and employing efficient algorithms, compute efficiency speeds up both model training and inference processes, ultimately cutting down on the time and resources needed, even when working with complex or large-scale models.

Efficient computation enables models to handle large datasets more effectively, minimizing bottlenecks associated with memory or processing power. Techniques such as parallel processing, hardware accelerators (e.g., GPUs, TPUs), and energy-aware scheduling contribute to reducing overhead while ensuring peak performance. As a result, compute efficiency not only supports model optimization but also enhances data handling, making it feasible to train models on high-quality datasets without unnecessary computational strain.

Edge Deployment Example. Edge deployments, such as those in autonomous vehicles, highlight the intricate balance required between real-time constraints and energy efficiency. Compute efficiency is central, as vehicles rely on high-performance onboard hardware to process massive streams of sensor data, including data from cameras, LiDAR, and radar, in real time. These computations must be performed with minimal latency to ensure safe navigation and split-second decision-making.

This compute efficiency is closely supported by algorithmic efficiency, as the system depends on compact, high-accuracy models designed for low latency. By employing streamlined neural network architectures or hybrid models combining deep learning and traditional algorithms, the computational demands on hardware are reduced. These optimized models not only lower the processing load but also consume less energy, reinforcing the system's overall energy efficiency.

Data efficiency enhances both compute and algorithmic efficiency by reducing the dependency on vast amounts of training data. Through synthetic and augmented datasets, the model can generalize effectively across diverse scenarios, including varying lighting, weather, and traffic conditions, without requiring extensive retraining. This targeted approach minimizes computational costs during training and allows the model to remain efficient while adapting to a wide range of real-world environments.

Together, the interdependence of these efficiencies ensures that autonomous vehicles can operate safely and reliably while minimizing energy consumption. This balance not only improves real-time performance but also contributes to broader goals, such as reducing fuel consumption and enhancing environmental sustainability.

9.4.2.3 Data Efficiency Strengthens Model and Compute

Data efficiency is fundamental to bolstering both model and compute efficiency. By focusing on high-quality, compact datasets, the training process becomes more streamlined, requiring fewer computational resources to achieve comparable or superior model performance. This targeted approach reduces data redundancy and minimizes the overhead associated with handling excessively large datasets.

Furthermore, data efficiency enables more focused model design. When datasets emphasize relevant features and minimize noise, models can achieve high performance with simpler architectures. Consequently, this reduces computational requirements during both training and inference, allowing more efficient use of computing resources.

Cloud Deployment Example. Cloud deployments exemplify how system efficiency can be achieved across interconnected dimensions. Consider a recommendation system operating in a data center, where high throughput and rapid inference are critical. Compute efficiency is achieved by leveraging parallelized processing on GPUs or TPUs, which optimize the computational workload to ensure timely and resource-efficient performance. This high-performance hardware allows the system to handle millions of simultaneous queries while keeping energy and operational costs in check.

This compute efficiency is bolstered by algorithmic efficiency, as the recommendation system employs streamlined architectures, such as pruned or simplified models. By reducing the computational and memory footprint, these models enable the system to scale efficiently, processing large volumes of data without overwhelming the infrastructure. The streamlined design also reduces the burden on accelerators, improving energy usage and maintaining throughput.

Data efficiency strengthens both compute and algorithmic efficiency by enabling the system to learn and adapt without excessive data overhead. By focusing on actively labeled datasets, the system can prioritize high-value training data, ensuring better model performance with fewer computational resources. This targeted approach reduces the size and complexity of training tasks, freeing up resources for inference and scaling while maintaining high recommendation accuracy.

Together, the interdependence of these efficiencies enables cloud-based systems to achieve a balance of performance, scalability, and cost-effectiveness. By optimizing model, compute, and data dimensions in harmony, cloud deployments become a cornerstone of modern AI applications, supporting millions of users with efficiency and reliability.

9.4.2.4 Efficiency Trade-offs

In many machine learning applications, efficiency is not merely a goal for optimization but a prerequisite for system feasibility. Extreme resource constraints, such as limited computational power, energy availability, and storage capacity, demand careful trade-offs between algorithmic efficiency, compute efficiency, and data efficiency. These constraints are particularly relevant in scenarios where machine learning models must operate in low-power embedded devices, remote sensors, or battery-operated systems.

Unlike cloud-based or even edge-based deployments, where computational resources are relatively abundant, resource-constrained environments require severe optimizations to ensure that models can function within tight operational limits. Achieving efficiency in such settings often involves trade-offs: smaller models may sacrifice some predictive accuracy, lower precision computations may introduce noise, and constrained datasets may limit generalization. The key challenge is to balance these trade-offs to maintain functionality while staying within strict power and compute budgets.

Tiny Deployment Case Study. A clear example of these trade-offs can be seen in Tiny ML, where machine learning models are deployed on ultra-low-power microcontrollers, often operating on milliwatts of power. Consider an IoT-based environmental monitoring system designed to detect temperature anomalies in remote agricultural fields. The device must process sensor data locally while operating on a small battery for months or even years without requiring recharging or maintenance.

In this setting, compute efficiency is critical, as the microcontroller has extremely limited processing capabilities, meaning the model must perform inference with minimal computational overhead. Algorithmic efficiency plays a central role, as the model must be compact enough to fit within the tiny memory available on the device, requiring streamlined architectures that eliminate unnecessary complexity. Data efficiency becomes essential, since collecting and storing large datasets in a remote location is impractical, requiring the model to learn effectively from small, carefully selected datasets to make reliable predictions with minimal training data.

Because of these constraints, Tiny ML deployments require a holistic approach to efficiency, where improvements in one area must compensate for limitations in another. A model that is computationally lightweight but requires excessive amounts of training data may not be viable. Similarly, a highly accurate model that demands too much energy will drain the battery too quickly. The success of Tiny ML hinges on balancing these interdependencies, ensuring that machine learning remains practical even in environments with severe resource constraints.

9.4.2.5 Progression and Takeaways

Starting with Mobile ML deployments and progressing to Edge ML, Cloud ML, and Tiny ML, these examples illustrate how system efficiency adapts to diverse operational contexts. Mobile ML emphasizes battery life and hardware limitations, edge systems balance real-time demands with energy efficiency, cloud systems prioritize scalability and throughput, and Tiny ML demonstrates how AI can thrive in environments with severe resource constraints.

Despite these differences, the fundamental principles remain consistent: achieving system efficiency requires optimizing model, compute, and data dimensions. These dimensions are deeply interconnected, with improvements in one often reinforcing the others. For instance, lightweight models enhance computational performance and reduce data requirements, while efficient hardware accelerates model training and inference. Similarly, focused datasets streamline model training and reduce computational overhead.

By understanding the interplay between these dimensions, we can design machine learning systems that meet specific deployment requirements while maintaining flexibility across contexts. For instance, a model architected for edge deployment can often be adapted for cloud scaling or simplified for mobile use, provided we carefully consider the relationships between model architecture, computational resources, and data requirements.

9.4.3 Scalability and Sustainability

System efficiency serves as a fundamental driver of environmental sustainability in machine learning systems. When systems are optimized for efficiency, they can be deployed at scale while minimizing their environmental footprint. This relationship creates a positive feedback loop, as sustainable design practices naturally encourage further efficiency improvements.

The interconnection between efficiency, scalability, and sustainability forms a virtuous cycle, as shown in Figure 9.15, that enhances the broader impact of machine learning systems. Efficient system design enables widespread deployment, which amplifies the positive environmental effects of sustainable practices. As organizations prioritize sustainability, they drive innovation in efficient system design, ensuring that advances in artificial intelligence align with global sustainability goals.

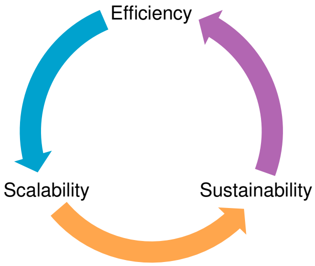


Figure 9.15: The virtuous cycle of machine learning system. Efficiency drives scalability and widespread adoption, which in turn drives the need for sustainable solutions, fueling the need for further efficiency.

9.4.3.1 Efficiency-Scalability Relationship

Efficient systems are inherently scalable. Reducing resource demands through lightweight models, targeted datasets, and optimized compute utilization allows systems to deploy broadly across diverse environments. For example, a speech recognition model that is efficient enough to run on mobile devices can serve millions of users globally without relying on costly infrastructure upgrades. Similarly, Tiny ML technologies, designed to operate on low-power hardware, make it possible to deploy thousands of devices in remote areas for applications like environmental monitoring or precision agriculture.

Scalability becomes feasible because efficiency reduces barriers to entry. Systems that are compact and energy-efficient require less infrastructure, making them more adaptable to different deployment contexts, from cloud data centers to edge and IoT devices. This adaptability is key to ensuring that advanced AI solutions reach users worldwide, fostering inclusion and innovation.

9.4.3.2 Scalability-Sustainability Relationship

When efficient systems scale, they amplify their contribution to sustainability. Energy-efficient designs deployed at scale reduce overall energy consumption and computational waste, mitigating the environmental impact of machine learning systems. For instance, deploying Tiny ML devices for on-device data processing avoids the energy costs of transmitting raw data to the cloud, while efficient recommendation engines in the cloud reduce the operational footprint of serving millions of users.

The wide-scale adoption of efficient systems not only reduces environmental costs but also fosters sustainable development in underserved regions. Efficient AI applications in healthcare, education, and agriculture can provide transformative benefits without imposing significant resource demands, aligning technological growth with ethical and environmental goals.

9.4.3.3 Sustainability-Efficiency Relationship

Sustainability itself reinforces the need for efficiency, creating a feedback loop that strengthens the entire system. Practices like minimizing data redundancy, designing energy-efficient hardware, and developing low-power models all emphasize efficient resource utilization. These efforts not only reduce the environmental footprint of AI systems but also set the stage for further scalability by making systems cost-effective and accessible.

9.5 Efficiency Trade-offs and Challenges

Thus far, we explored how the dimensions of system efficiency, including algorithmic efficiency, compute efficiency, and data efficiency, are deeply interconnected. Ideally, these dimensions reinforce one another, creating a system that is both efficient and high-performing. Compact models reduce computational demands, efficient hardware accelerates processes, and high-quality datasets streamline training and inference. However, achieving this harmony is far from straightforward.

9.5.1 Trade-offs Source

In practice, balancing these dimensions often uncovers underlying tensions. Improvements in one area can impose constraints on others, highlighting the interconnected nature of machine learning systems. For instance, simplifying a model to reduce computational demands might result in reduced accuracy, while optimizing compute efficiency for real-time responsiveness can conflict with energy efficiency goals. These trade-offs are not limitations but reflections of the intricate design decisions required to build adaptable and efficient systems.

Understanding the root of these trade-offs is essential for navigating the challenges of system design. Each efficiency dimension influences the others, creating a dynamic interplay that shapes system performance. The following sections delve into these interdependencies, beginning with the relationship between algorithmic efficiency and compute requirements.

9.5.1.1 Efficiency and Compute Requirements

Model efficiency focuses on designing compact and streamlined models that minimize computational and memory demands. By reducing the size or complexity of a model, it becomes easier to deploy on devices with limited resources, such as mobile phones or IoT sensors.

However, overly simplifying a model can reduce its accuracy, especially for complex tasks. To make up for this loss, additional computational resources may be required during training to fine-tune the model or during deployment to apply more sophisticated inference algorithms. Thus, while algorithmic efficiency can reduce computational costs, achieving this often places additional strain on compute efficiency.

9.5.1.2 Efficiency and Real-Time Needs

Compute efficiency aims to minimize the resources required for tasks like training and inference, reducing energy consumption, processing time, and memory use. In many applications, particularly in cloud computing or data centers, this optimization works seamlessly with algorithmic efficiency to improve system performance.

However, in scenarios that require real-time responsiveness, including autonomous vehicles and augmented reality, compute efficiency is harder to maintain. Figure 9.16 illustrates this challenge: real-time systems often require high-performance hardware to process large amounts of data instantly, which can conflict with energy efficiency goals or increase system costs. Balancing compute efficiency with stringent real-time application needs becomes a key challenge in such applications.

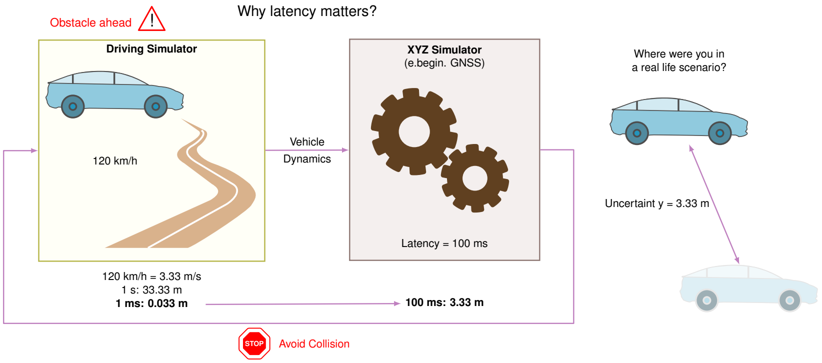


Figure 9.16: An example in the autonomous vehicle (AV) setting, where both efficiency and latency both matter. One cannot easily increase compute to drive down latency, but latency cannot be sacrified as it might impact safety, where increase processing time might impact reation time and braking distance.

9.5.1.3 Efficiency and Model Generalization

Data efficiency seeks to minimize the amount of data required to train a model without sacrificing performance. By curating smaller, high-quality datasets, the training process becomes faster and less resource-intensive. Ideally, this reinforces both model and compute efficiency, as smaller datasets reduce the computational load and support more compact models.

However, reducing the size of a dataset can also limit its diversity, making it harder for the model to generalize to unseen scenarios. To address this, additional compute resources or model complexity may be required, creating a tension between data efficiency and the broader goals of system efficiency.

9.5.1.4 Summary

The interdependencies between model, compute, and data efficiency are the foundation of a well-designed machine learning system. While these dimensions can reinforce one another, building a system that achieves this synergy often requires navigating difficult trade-offs. These trade-offs highlight the complexity of designing machine learning systems that balance performance, scalability, and resource constraints.

9.5.2 Common Trade-offs

In machine learning system design, trade-offs are an inherent reality. As we explored in the previous section, the interdependencies between algorithmic efficiency, compute efficiency, and data efficiency ideally work together to create powerful, resource-conscious systems. However, achieving this harmony is far from straightforward. In practice, improvements in one dimension often come at the expense of another. Designers must carefully weigh these trade-offs to achieve a balance that aligns with the system's goals and deployment context.

This balancing act is especially challenging because trade-offs are rarely one-dimensional. Decisions made in one area often have cascading effects on the rest of the system. For instance, choosing a larger, more complex model may improve accuracy, but it also increases computational demands and the size of the training dataset required. Similarly, reducing energy consumption may limit the ability to meet real-time performance requirements, particularly in latency-sensitive applications.

We explore three of the most common trade-offs encountered in machine learning system design:

1. **Model complexity vs. compute resources**,
2. **Energy efficiency vs. real-time performance**, and
3. **Data size vs. model generalization**.

Each of these trade-offs illustrates the nuanced decisions that system designers must make and the challenges involved in achieving efficient, high-performing systems.

9.5.2.1 Complexity vs. Resources

The relationship between model complexity and compute resources is one of the most fundamental trade-offs in machine learning system design. Complex models, such as deep neural networks with millions or even billions of parameters, are often capable of achieving higher accuracy by capturing intricate patterns in data. However, this complexity comes at a cost. These models require significant computational power and memory to train and deploy, often making them impractical for environments with limited resources.

For example, consider a recommendation system deployed in a cloud data center. A highly complex model may deliver better recommendations, but it increases the computational demands on servers, leading to higher energy consumption and operating costs. On the other hand, a simplified model may reduce these demands but might compromise the quality of recommendations, especially when handling diverse or unpredictable user behavior.

The trade-off becomes even more pronounced in resource-constrained environments such as mobile or edge devices. A compact, streamlined model designed for a smartphone or an autonomous vehicle may operate efficiently within the device's hardware limits but might require more sophisticated data preprocessing or training procedures to compensate for its reduced capacity. This balancing act highlights the interconnected nature of efficiency dimensions, where gains in one area often demand sacrifices in another.

9.5.2.2 Energy vs. Performance

Energy efficiency and real-time performance often pull machine learning systems in opposite directions, particularly in applications requiring low-latency responses. Real-time systems, such as those in autonomous vehicles or augmented reality applications, rely on high-performance hardware to process large volumes of data quickly. This ensures responsiveness and safety in scenarios where even small delays can lead to significant consequences. However, achieving such performance typically increases energy consumption, creating tension with the goal of minimizing resource use.

For instance, an autonomous vehicle must process sensor data from cameras, LiDAR, and radar in real time to make navigation decisions. The computational demands of these tasks often require specialized accelerators, such as GPUs, which can consume significant energy. While optimizing hardware utilization and model architecture can improve energy efficiency to some extent, the demands of real-time responsiveness make it challenging to achieve both goals simultaneously.

In edge deployments, where devices rely on battery power or limited energy sources, this trade-off becomes even more critical. Striking a balance between energy efficiency and real-time performance often involves prioritizing one over the other, depending on the application's requirements. This trade-off underscores the importance of context-specific design, where the constraints and priorities of the deployment environment dictate the balance between competing objectives.

9.5.2.3 Data Size vs. Generalization

The size and quality of the dataset used to train a machine learning model play a role in its ability to generalize to new, unseen data. Larger datasets generally provide greater diversity and coverage, enabling models to capture subtle patterns and reduce the risk of overfitting. However, the computational and memory demands of training on large datasets can be substantial, leading to trade-offs between data efficiency and computational requirements.

In resource-constrained environments such as Tiny ML deployments, the challenge of dataset size is particularly evident. For example, an IoT device monitoring environmental conditions might need a model that generalizes well to varying temperatures, humidity levels, or geographic regions. Collecting and processing extensive datasets to capture these variations may be impractical due to storage, computational, and energy limitations. In such cases, smaller, carefully curated datasets or synthetic data generated to mimic real-world conditions are used to reduce computational strain. However, this reduction often risks missing key edge cases, which could degrade the model's performance in diverse environments.

Conversely, in cloud-based systems, where compute resources are more abundant, training on massive datasets can still pose challenges. Managing data redundancy, ensuring high-quality labeling, and handling the time and cost associated with large-scale data pipelines often require significant computational infrastructure. This trade-off highlights how the need to balance dataset size and model generalization depends heavily on the deployment context and available resources.

9.5.2.4 Summary

The interplay between model complexity, compute resources, energy efficiency, real-time performance, and dataset size illustrates the inherent trade-offs in machine learning system design. These trade-offs are rarely one-dimensional; decisions to optimize one aspect of a system often ripple through the others, requiring careful consideration of the specific goals and constraints of the application.

Designers must weigh the advantages and limitations of each trade-off in the context of the deployment environment. For instance, a cloud-based system might prioritize scalability and throughput over energy efficiency, while an edge system must balance real-time performance with strict power constraints. Similarly, resource-limited Tiny ML deployments require exceptional data and algorithmic efficiency to operate within severe hardware restrictions.

By understanding these common trade-offs, we can begin to identify strategies for navigating them effectively. The next section will explore practical approaches to managing these tensions, focusing on techniques and design principles that enable system efficiency while addressing the complexities of real-world applications.

9.6 Managing Trade-offs

The trade-offs inherent in machine learning system design require thoughtful strategies to navigate effectively. While the interdependencies between algorithmic efficiency, compute efficiency, and data efficiency create opportunities for synergy, achieving this balance often involves difficult decisions. The specific goals and constraints of the deployment environment heavily influence how these trade-offs are addressed. For example, a system designed for cloud deployment may prioritize scalability and throughput, while a Tiny ML system must focus on extreme resource efficiency.

To manage these challenges, designers can adopt a range of strategies that address the unique requirements of different contexts. By prioritizing efficiency dimensions based on the application, collaborating across system components, and leveraging automated optimization tools, it is possible to create systems that balance performance, cost, and resource use. This section explores these approaches and provides guidance for designing systems that are both efficient and adaptable.

9.6.1 Contextual Prioritization

Efficiency goals are rarely universal. The specific demands of an application or deployment scenario heavily influence which dimension of efficiency, whether it be model, compute, or data, takes precedence. Designing an efficient system requires a deep understanding of the operating environment and the constraints it imposes. Prioritizing the right dimensions based on context is the first step in effectively managing trade-offs.

For instance, in Mobile ML deployments, battery life is often the primary constraint. This places a premium on compute efficiency, as energy consumption must be minimized to preserve the device's operational time. As a result, lightweight models are prioritized, even if it means sacrificing some accuracy or requiring additional data preprocessing. The focus is on balancing acceptable performance with energy-efficient operation.

In contrast, Cloud ML-based systems prioritize scalability and throughput. These systems must process large volumes of data and serve millions of users simultaneously. While compute resources in cloud environments are more abundant, energy efficiency and operational costs still remain important considerations. Here, algorithmic efficiency plays a critical role in ensuring that the system can scale without overwhelming the underlying infrastructure.

Edge ML systems present an entirely different set of priorities. Autonomous vehicles or real-time monitoring systems require low-latency processing to ensure safe and reliable operation. This makes real-time performance and compute efficiency paramount, often at the expense of energy consumption. However, the hardware constraints of edge devices mean that these systems must still carefully manage energy and computational resources to remain viable.

Finally, Tiny ML deployments demand extreme levels of efficiency due to the severe limitations of hardware and energy availability. For these systems, model and data efficiency are the top priorities. Models must be highly compact and capable of operating on microcontrollers with minimal memory and compute power. At the same time, the training process must rely on small, carefully curated datasets to ensure the model generalizes well without requiring extensive resources.

In each of these contexts, prioritizing the right dimensions of efficiency ensures that the system meets its functional and resource requirements. Recognizing the unique demands of each deployment scenario allows designers to navigate trade-offs effectively and tailor solutions to specific needs.

9.6.2 Test-Time Compute

We can further enhance system adaptability through dynamic resource allocation during inference, a concept often referred to as "Test-Time Compute." This approach recognizes that resource needs may fluctuate even within a specific deployment context. By adjusting the computational effort expended at inference time, systems can fine-tune their performance to meet immediate demands.

For example, in a cloud-based video analysis system, standard video streams might be processed with a streamlined, low-compute model to maintain high throughput. However, when a critical event is detected, the system could dynamically allocate more computational resources to a more complex model, enabling higher precision analysis of the event. This flexibility allows for a trade-off between latency and accuracy on demand.

Similarly, in mobile applications, a voice assistant might use a lightweight model for routine commands, conserving battery life. But when faced with a complex query, the system could temporarily activate a more resource-intensive model for improved accuracy. This ability to adjust compute based on the complexity of the task or the importance of the result is a powerful tool for optimizing system performance in real-time.

However, implementing "Test-Time Compute" introduces new challenges. Dynamic resource allocation requires sophisticated monitoring and control mechanisms to ensure that the system remains stable and responsive. Additionally, there is a point of diminishing returns; increasing compute beyond a certain threshold may not yield significant performance improvements, making it crucial to strike a balance between resource usage and desired outcomes. Furthermore, the ability to dynamically increase compute can create disparities in access to high-performance AI, raising equity concerns about who benefits from advanced AI capabilities.

Despite these challenges, "Test-Time Compute" offers a valuable strategy for enhancing system adaptability and optimizing performance in dynamic environments. It complements the contextual prioritization approach by enabling systems to respond effectively to varying demands within specific deployment scenarios.

9.6.3 Co-Design

Efficient machine learning systems are rarely the product of isolated optimizations. Achieving balance across model, compute, and data efficiency requires an end-to-end perspective, where each component of the system is designed in tandem with the others. This holistic approach, often referred to as co-design, involves aligning model architectures, hardware platforms, and data pipelines to work seamlessly together.

One of the key benefits of co-design is its ability to mitigate trade-offs by tailoring each component to the specific requirements of the system. For instance, consider a speech recognition system deployed on a mobile device. The model must be compact enough to fit within the device's tiny ML memory constraints while still delivering real-time performance. By designing the model architecture to leverage the capabilities of hardware accelerators, such as NPUs, it becomes possible to achieve low-latency inference without excessive energy consumption. Similarly, careful preprocessing and augmentation of the training data can ensure robust performance, even with a smaller, streamlined model.

Co-design becomes essential in resource-constrained environments like Edge ML and Tiny ML deployments. Models must align precisely with hardware capabilities. For example, 8-bit models[13] require hardware support for efficient integer operations, while pruned models benefit from sparse tensor operations. Similarly, edge accelerators often optimize specific operations like convolutions

[13] 8-bit models: ML models use 8-bit integer representations for weights and activations instead of the standard 32-bit floating-point format, reducing memory usage and computational requirements for faster, more energy-efficient inference on compatible hardware.

or matrix multiplication, influencing model architecture choices. This creates a tight coupling between hardware and model design decisions.

This approach extends beyond the interaction of models and hardware. Data pipelines, too, play a central role in co-design. For example, in applications requiring real-time adaptation, such as personalized recommendation systems, the data pipeline must deliver high-quality, timely information that minimizes computational overhead while maximizing model effectiveness. By integrating data management into the design process, it becomes possible to reduce redundancy, streamline training, and support efficient deployment.

End-to-end co-design ensures that the trade-offs inherent in machine learning systems are addressed holistically. By designing each component with the others in mind, it becomes possible to balance competing priorities and create systems that are not only efficient but also robust and adaptable.

9.6.4 Automation

Navigating the trade-offs between model, compute, and data efficiency is a complex task that often involves numerous iterations and expert judgment. Automation and optimization tools have emerged as powerful solutions for managing these challenges, streamlining the process of balancing efficiency dimensions while reducing the time and expertise required.

One widely used approach is automated machine learning (AutoML), which enables the exploration of different model architectures, hyperparameter configurations, and feature engineering techniques. By automating these aspects of the design process, AutoML can identify models that achieve an optimal balance between performance and efficiency. For instance, an AutoML pipeline might search for a lightweight model architecture that delivers high accuracy while fitting within the resource constraints of an edge device (F. Hutter, Kotthoff, and Vanschoren 2019a). This approach reduces the need for manual trial-and-error, making optimization faster and more accessible.

Neural architecture search (NAS) takes automation a step further by designing model architectures tailored to specific hardware or deployment scenarios. NAS algorithms evaluate a wide range of architectural possibilities, selecting those that maximize performance while minimizing computational demands. For example, NAS can design models that leverage quantization or sparsity techniques, ensuring compatibility with energy-efficient accelerators like TPUs or microcontrollers (Elsken, Metzen, and Hutter 2019a). This automated co-design of models and hardware helps mitigate trade-offs by aligning efficiency goals across dimensions.

Data efficiency, too, benefits from automation. Tools that automate dataset curation, augmentation, and active learning reduce the size of training datasets without sacrificing model performance. These tools prioritize high-value data points, ensuring that models are trained on the most informative examples. This not only speeds up training but also reduces computational overhead, reinforcing both compute and algorithmic efficiency (Settles 2012b).

While automation tools are not a panacea, they play a critical role in addressing the complexity of trade-offs. By leveraging these tools, system designers can achieve efficient solutions more quickly and at lower cost, freeing them to focus on broader design challenges and deployment considerations.

9.6.5 Summary

Designing efficient machine learning systems requires a deliberate approach to managing trade-offs between model, compute, and data efficiency. These trade-offs are influenced by the context of the deployment, the constraints of the hardware, and the goals of the application. By prioritizing efficiency dimensions based on the specific needs of the system, embracing end-to-end co-design, and leveraging automation tools, it becomes possible to navigate these challenges effectively.

The strategies explored illustrate how thoughtful design can transform trade-offs into opportunities for synergy. For example, aligning model architectures with hardware capabilities can mitigate energy constraints, while automation tools like AutoML and NAS streamline the process of optimizing efficiency dimensions. These approaches underscore the importance of treating system efficiency as a holistic endeavor, where components are designed to complement and reinforce one another.

9.7 Efficiency-First Mindset

Designing an efficient machine learning system requires a holistic approach. While it is tempting to focus on optimizing individual components, such as the model architecture or the hardware platform, true efficiency emerges when the entire system is considered as a whole. This end-to-end perspective ensures that trade-offs are balanced across all stages of the machine learning pipeline, from data collection to deployment.

Efficiency is not a static goal but a dynamic process shaped by the context of the application. A system designed for a cloud data center will prioritize scalability and throughput, while an edge deployment will focus on low latency and energy conservation. These differing priorities influence decisions at every step of the design process, requiring careful alignment of the model, compute resources, and data strategy.

An end-to-end perspective can transform system design, enabling machine learning practitioners to build systems that effectively balance trade-offs. Through case studies and examples, we will highlight how efficient systems are designed to meet the unique challenges of their deployment environments, whether in the cloud, on mobile devices, or in resource-constrained Tiny ML applications.

9.7.1 End-to-End Perspective

Efficiency in machine learning systems is achieved not through isolated optimizations but by considering the entire pipeline as a unified whole. Each stage, including data collection, model training, hardware deployment, and inference, contributes to the overall efficiency of the system. Decisions made at one stage can ripple through the rest, influencing performance, resource use, and scalability.

For example, data collection and preprocessing are often the starting points of the pipeline. The quality and diversity of the data directly impact model performance and efficiency. Curating smaller, high-quality datasets can reduce computational costs during training while simplifying the model's design. However, insufficient data diversity may affect generalization, necessitating compensatory measures in model architecture or training procedures. By aligning the data strategy with the model and deployment context, designers can avoid inefficiencies downstream.

Model training is another critical stage. The choice of architecture, optimization techniques, and hyperparameters must consider the constraints of the deployment hardware. A model designed for high-performance cloud systems may emphasize accuracy and scalability, leveraging large datasets and compute resources. Conversely, a model intended for edge devices must balance accuracy with size and energy efficiency, often requiring compact architectures and quantization techniques tailored to specific hardware.

Deployment and inference demand precise hardware alignment. Each platform offers distinct capabilities. GPUs excel at parallel matrix operations, TPUs optimize specific neural network computations, and microcontrollers provide energy-efficient scalar processing. For example, a smartphone speech recognition system might leverage an NPU's dedicated convolution units for 5-millisecond inference times at 1-watt power draw, while an autonomous vehicle's FPGA-based accelerator processes multiple sensor streams with 50-microsecond latency. This hardware-software integration determines real-world efficiency.

An end-to-end perspective ensures that trade-offs are addressed holistically, rather than shifting inefficiencies from one stage of the pipeline to another. By treating the system as an integrated whole, machine learning practitioners can design solutions that are not only efficient but also robust and scalable across diverse deployment scenarios.

9.7.2 Scenarios

The efficiency needs of machine learning systems differ significantly depending on the lifecycle stage and deployment environment. From research prototypes to production systems, and from high-performance cloud applications to resource-constrained edge deployments, each scenario presents unique challenges and trade-offs. Understanding these differences is crucial for designing systems that meet their operational requirements effectively.

9.7.2.1 Prototypes vs. Production

In the research phase, the primary focus is often on model performance, with efficiency taking a secondary role. Prototypes are typically trained and tested using abundant compute resources, allowing researchers to experiment with large architectures, extensive hyperparameter tuning, and diverse datasets. While this approach enables the exploration of cutting-edge techniques, the resulting systems are often too resource-intensive for real-world use.

In contrast, production systems must prioritize efficiency to operate within practical constraints. Deployment environments, including cloud data centers, mobile devices, and IoT sensors, impose strict limitations on compute power, memory, and energy consumption. Transitioning from a research prototype to a production-ready system often involves significant optimization, such as model pruning, quantization, or retraining on targeted datasets. This shift highlights the need to balance performance and efficiency as systems move from concept to deployment.

9.7.2.2 Cloud Apps vs. Constrained Systems

Cloud-based systems, such as those used for large-scale analytics or recommendation engines, are designed to handle massive workloads. Scalability is the primary concern, requiring models and infrastructure that can support millions of users simultaneously. While compute resources are relatively abundant in cloud environments, energy efficiency and operational costs remain critical considerations. Techniques such as model compression and hardware-specific optimizations help manage these trade-offs, ensuring the system scales efficiently.

In contrast, edge and mobile systems operate under far stricter constraints. Real-time performance, energy efficiency, and hardware limitations are often the dominant concerns. For example, a speech recognition application on a smartphone must balance model size and latency to provide a seamless user experience without draining the device's battery. Similarly, an IoT sensor deployed in a remote location must operate for months on limited power, requiring an ultra-efficient model and compute pipeline. These scenarios demand solutions that prioritize efficiency over raw performance.

9.7.2.3 Frequent Retraining vs. Stability

Some systems, such as recommendation engines or fraud detection platforms, require frequent retraining to remain effective in dynamic environments. These systems depend heavily on data efficiency, using actively labeled datasets and sampling strategies to minimize retraining costs. Compute efficiency also plays a role, as scalable infrastructure is needed to process new data and update models regularly.

Other systems, such as embedded models in medical devices or industrial equipment, require long-term stability with minimal updates. In these cases, upfront optimizations in model and data efficiency are critical to ensure the system performs reliably over time. Reducing dependency on frequent updates minimizes computational and operational overhead, making the system more sustainable in the long run.

9.7.3 Summary

Designing machine learning systems with efficiency in mind requires a holistic approach that considers the specific needs and constraints of the deployment context. From research prototypes to production systems, and across environments as varied as cloud data centers, mobile devices, and Tiny ML applications, the priorities for efficiency differ significantly. Each stage of the machine learning pipeline, including data collection, model design, training, deployment, and inference, presents unique trade-offs that must be navigated thoughtfully.

The examples and scenarios in this section demonstrate the importance of aligning system design with operational requirements. Cloud systems prioritize scalability and throughput, edge systems focus on real-time performance, and Tiny ML applications emphasize extreme resource efficiency. Understanding these differences enables practitioners to tailor their approach, leveraging strategies such as end-to-end co-design and automation tools to balance competing priorities effectively.

Ultimately, the key to designing efficient systems lies in recognizing that efficiency is not a one-size-fits-all solution. It is a dynamic process that requires careful consideration of trade-offs, informed prioritization, and a commitment to addressing the unique challenges of each scenario. With these principles in mind, machine learning practitioners can create systems that are not only efficient but also robust, scalable, and sustainable.

9.8 Broader Challenges

While efficiency in machine learning is often framed as a technical challenge, it is also deeply tied to broader questions about the purpose and impact of AI systems. Designing efficient systems involves navigating not only practical trade-offs but also complex ethical and philosophical considerations, such as the following:

- What are the limits of optimization?
- How do we ensure that efficiency benefits are distributed equitably?
- Can the pursuit of efficiency stifle innovation or creativity in the field?

We must explore these questions as engineers, inviting reflection on the broader implications of system efficiency. By examining the limits of optimization, equity concerns, and the tension between innovation and efficiency, we can have a deeper understanding of the challenges involved in balancing technical goals with ethical and societal values.

9.8.1 Optimization Limits

Optimization plays a central role in building efficient machine learning systems, but it is not an infinite process. As systems become more refined, each additional improvement often requires exponentially more effort, time, or resources, while delivering increasingly smaller benefits. This phenomenon, known as diminishing returns, is a common challenge in many engineering domains, including machine learning.

The No Free Lunch (NFL) theorems for optimization further illustrate the inherent limitations of optimization efforts. According to the NFL theorems, no single optimization algorithm can outperform all others across every possible problem. This implies that the effectiveness of an optimization technique is highly problem-specific, and improvements in one area may not translate to others (Wolpert and Macready 1997).

For example, compressing a machine learning model can initially reduce memory usage and compute requirements significantly with minimal loss in accuracy. However, as compression progresses, maintaining performance becomes increasingly challenging. Achieving additional gains may necessitate sophisticated techniques, such as hardware-specific optimizations or extensive retraining, which increase both complexity and cost. These costs extend beyond financial investment in specialized hardware and training resources to include the time and expertise required to fine-tune models, iterative testing efforts, and potential trade-offs in model robustness and generalizability. As such, pursuing extreme efficiency often leads to diminishing returns, where escalating costs and complexity outweigh incremental benefits.

The NFL theorems highlight that no universal optimization solution exists, emphasizing the need to balance efficiency pursuits with practical considerations. Recognizing the limits of optimization is critical for designing systems that are not only efficient but also practical and sustainable. Over-optimization risks wasted resources and reduced adaptability, complicating future system updates or adjustments to changing requirements. Identifying when a system is "good enough" ensures resources are allocated effectively, focusing on efforts with the greatest overall impact.

Similarly, optimizing datasets for training efficiency may initially save resources but excessively reducing dataset size risks compromising diversity and weakening model generalization. Likewise, pushing hardware to its performance limits may improve metrics such as latency or power consumption, yet the associated reliability concerns and engineering costs can ultimately outweigh these gains.

In summary, understanding the limits of optimization is essential for creating systems that balance efficiency with practicality and sustainability. This perspective helps avoid over-optimization and ensures resources are invested in areas with the most meaningful returns.

9.8.2 Moore's Law Case Study

One of the most insightful examples of the limits of optimization can be seen in Moore's Law and the economic curve it depends on. While Moore's Law is often celebrated as a predictor of exponential growth in computational power, its success relied on an intricate economic balance. The relationship between integration and cost, as illustrated in the accompanying plot, provides a compelling analogy for the diminishing returns seen in machine learning optimization.

Figure 9.17 shows the relative manufacturing cost per component as the number of components in an integrated circuit increases. Initially, as more components are packed onto a chip (x-axis), the cost per component (y-axis) decreases. This is because higher integration reduces the need for supporting infrastructure such as packaging and interconnects, creating economies of scale. For example, in the early years of integrated circuit design, moving from hundreds to thousands of components per chip drastically reduced costs and improved performance (Moore 2021).

However, as integration continues, the curve begins to rise. This inflection point occurs because the challenges of scaling become more pronounced. Components packed closer together face reliability issues, such as increased heat dissipation and signal interference. Addressing these issues requires more sophisticated manufacturing techniques, such as advanced lithography, error

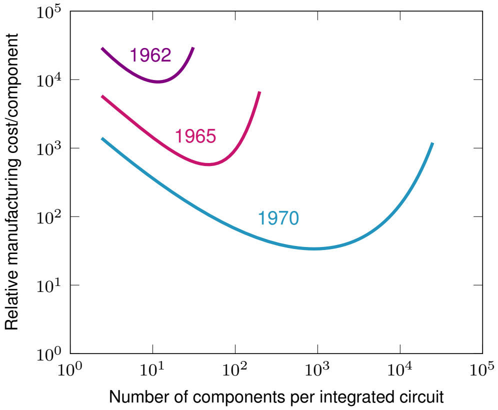


Figure 9.17: The economics of Moore's law. Source: (Moore 2021)

correction, and improved materials. These innovations increase the complexity and cost of production, driving the curve upward. This U-shaped curve captures the fundamental trade-off in optimization: early improvements yield substantial benefits, but beyond a certain point, each additional gain comes at a greater cost.

9.8.2.1 ML Optimization Parallels

The dynamics of this curve mirror the challenges faced in machine learning optimization. For instance, compressing a deep learning model to reduce its size and energy consumption follows a similar trajectory. Initial optimizations, such as pruning redundant parameters or reducing precision, often lead to significant savings with minimal impact on accuracy. However, as the model is further compressed, the losses in performance become harder to recover. Techniques such as quantization or hardware-specific tuning can restore some of this performance, but these methods add complexity and cost to the design process.

Similarly, in data efficiency, reducing the size of training datasets often improves computational efficiency at first, as less data requires fewer resources to process. Yet, as the dataset shrinks further, it may lose diversity, compromising the model's ability to generalize. Addressing this often involves introducing synthetic data or sophisticated augmentation techniques, which demand additional engineering effort.

The Moore's Law plot (Figure 9.17) serves as a visual reminder that optimization is not an infinite process. The cost-benefit balance is always context-dependent, and the point of diminishing returns varies based on the goals and constraints of the system. Machine learning practitioners, like semiconductor engineers, must identify when further optimization ceases to provide meaningful benefits. Over-optimization can lead to wasted resources, reduced adaptability, and systems that are overly specialized to their initial conditions.

9.8.3 Equity Concerns

Efficiency in machine learning has the potential to reduce costs, improve scalability, and expand accessibility. However, the resources needed to achieve efficiency, including advanced hardware, curated datasets, and state-of-the-art optimization techniques, are often concentrated in well-funded organizations or regions. This disparity creates inequities in who can leverage efficiency gains, limiting the reach of machine learning in low-resource contexts. By examining compute, data, and algorithmic efficiency inequities, we can better understand these challenges and explore pathways toward democratization.

9.8.3.1 Uneven Access

The training costs of state-of-the-art AI models have reached unprecedented levels. For example, OpenAI's GPT-4 used an estimated USD $78 million worth of compute to train, while Google's Gemini Ultra cost USD $191 million for compute (Maslej et al. 2024). Computational efficiency depends on access to specialized hardware and infrastructure. The discrepancy in access is significant: training even a small language model (SLM) like LLlama with 7 billion parameters can require millions of dollars in computing resources, while many research institutions operate with significantly lower annual compute budgets.

Research conducted by OECD.AI indicates that 90% of global AI computing capacity is centralized in only five countries, posing significant challenges for researchers and professionals in other regions (OECD.AI 2021).

A concrete illustration of this disparity is the compute divide in academia versus industry. Academic institutions often lack the hardware needed to replicate state-of-the-art results, particularly when competing with large technology firms that have access to custom supercomputers or cloud resources. This imbalance not only stifles innovation in underfunded sectors but also makes it harder for diverse voices to contribute to advancing machine learning.

Energy-efficient compute technologies, such as accelerators designed for Tiny ML or Mobile ML, present a promising avenue for democratization. By enabling powerful processing on low-cost, low-power devices, these technologies allow organizations without access to high-end infrastructure to build and deploy impactful systems. For instance, energy-efficient Tiny ML models can be deployed on affordable microcontrollers, opening doors for applications in healthcare, agriculture, and education in underserved regions.

9.8.3.2 Low-Resource Challenges

Data efficiency is essential in contexts where high-quality datasets are scarce, but the challenges of achieving it are unequally distributed. For example, natural language processing (NLP) for low-resource languages suffers from a lack of sufficient training data, leading to significant performance gaps compared to high-resource languages like English. Efforts like the Masakhane project, which builds open-source datasets for African languages, show how collaborative initiatives can address this issue. However, scaling such efforts globally requires far greater investment and coordination.

Even when data is available, the ability to process and curate it efficiently depends on computational and human resources. Large organizations routinely employ data engineering teams and automated pipelines for curation and augmentation, enabling them to optimize data efficiency and improve downstream performance. In contrast, smaller groups often lack access to the tools or expertise needed for such tasks, leaving them at a disadvantage in both research and practical applications.

Democratizing data efficiency requires more open sharing of pre-trained models and datasets. Initiatives like Hugging Face's open access to transformers or multilingual models by organizations like Meta's No Language Left Behind aim to make state-of-the-art NLP models available to researchers and practitioners worldwide. These efforts help reduce the barriers to entry for data-scarce regions, enabling more equitable access to AI capabilities.

9.8.3.3 Efficiency for Accessibility

Model efficiency plays a crucial role in democratizing machine learning by enabling advanced capabilities on low-cost, resource-constrained devices. Compact, efficient models designed for edge devices or mobile phones have already begun to bridge the gap in accessibility. For instance, AI-powered diagnostic tools running on smartphones are transforming healthcare in remote areas, while low-power Tiny ML models enable environmental monitoring in regions without reliable electricity or internet connectivity.

Technologies like TensorFlow Lite and PyTorch Mobile allow developers to deploy lightweight models on everyday devices, expanding access to AI applications in resource-constrained settings. These tools demonstrate how algorithmic efficiency can serve as a practical pathway to equity, particularly when combined with energy-efficient compute hardware.

However, scaling the benefits of algorithmic efficiency requires addressing barriers to entry. Many efficient architectures, such as those designed through NAS, remain resource-intensive to develop. Open-source efforts to share pre-optimized models, like MobileNet or EfficientNet, play a critical role in democratizing access to efficient AI by allowing under-resourced organizations to deploy state-of-the-art solutions without needing to invest in expensive optimization processes.

9.8.3.4 Democratization Pathways

Efforts to close the equity gap in machine learning must focus on democratizing access to tools and techniques that enhance efficiency. Open-source initiatives, such as community-driven datasets and shared model repositories, provide a foundation for equitable access to efficient systems. Affordable hardware platforms, such as Raspberry Pi devices or open-source microcontroller frameworks, further enable resource-constrained organizations to build and deploy AI solutions tailored to their needs.

Collaborative partnerships between well-resourced organizations and underrepresented groups also offer opportunities to share expertise, funding, and infrastructure. For example, initiatives that provide subsidized access to cloud computing platforms or pre-trained models for underserved regions can empower diverse communities to leverage efficiency for social impact.

Through efforts in model, computation, and data efficiency, the democratization of machine learning can become a reality. These efforts not only expand access to AI capabilities but also foster innovation and inclusivity, ensuring that the benefits of efficiency are shared across the global community.

9.8.4 Balancing Innovation and Efficiency

The pursuit of efficiency in machine learning often brings with it a tension between optimizing for what is known and exploring what is new. On one hand, efficiency drives the practical deployment of machine learning systems, enabling scalability, cost reduction, and environmental sustainability. On the other hand, focusing too heavily on efficiency can stifle innovation by discouraging experimentation with untested, resource-intensive ideas.

9.8.4.1 Stability vs. Experimentation

Efficiency often favors established techniques and systems that have already been proven to work well. For instance, optimizing neural networks through pruning, quantization, or distillation typically involves refining existing architectures rather than developing entirely new ones. While these approaches provide incremental improvements, they may come at the cost of exploring novel designs or paradigms that could yield transformative breakthroughs.

Consider the shift from traditional machine learning methods to deep learning. Early neural network research in the 1990s and 2000s required significant computational resources and often failed to outperform simpler methods on practical tasks. Despite this, researchers continued to push the boundaries of what was possible, eventually leading to the breakthroughs in deep learning that define modern AI. If the field had focused exclusively on efficiency during that period, these innovations might never have emerged.

9.8.4.2 Resource-Intensive Innovation

Pioneering research often requires significant resources, from massive datasets to custom hardware. For example, large language models like GPT-4 or PaLM are not inherently efficient; their training processes consume enormous amounts of compute power and energy. Yet, these models have opened up entirely new possibilities in language understanding, prompting advancements that eventually lead to more efficient systems, such as smaller fine-tuned versions for specific tasks.

However, this reliance on resource-intensive innovation raises questions about who gets to participate in these advancements. Well-funded organizations can afford to explore new frontiers, while smaller institutions may be constrained to incremental improvements that prioritize efficiency over novelty. Balancing the need for experimentation with the realities of resource availability is a key challenge for the field.

9.8.4.3 Efficiency-Creativity Constraint

Efficiency-focused design often requires adhering to strict constraints, such as reducing model size, energy consumption, or latency. While these constraints can drive ingenuity, they can also limit the scope of what researchers and engineers are willing to explore. For instance, edge computing applications often demand ultra-compact models, leading to a narrow focus on compression techniques rather than entirely new approaches to machine learning on constrained devices.

At the same time, the drive for efficiency can have a positive impact on innovation. Constraints force researchers to think creatively, leading to the development of new methods that maximize performance within tight resource budgets. Techniques like NAS and attention mechanisms arose, in part, from the need to balance performance and efficiency, demonstrating that innovation and efficiency can coexist when approached thoughtfully.

9.8.4.4 Striking a Balance

The tension between innovation and efficiency highlights the need for a balanced approach to system design and research priorities. Organizations and researchers must recognize when it is appropriate to prioritize efficiency and when to embrace the risks of experimentation. For instance, applied systems for real-world deployment may demand strict efficiency constraints, while exploratory research labs can focus on pushing boundaries without immediate concern for resource optimization.

Ultimately, the relationship between innovation and efficiency is not adversarial but complementary. Efficient systems create the foundation for scalable, practical applications, while resource-intensive experimentation drives the breakthroughs that redefine what is possible. Balancing these priorities ensures that machine learning continues to evolve while remaining accessible, impactful, and sustainable.

9.9 Conclusion

Efficiency in machine learning systems is essential not just for achieving technical goals but for addressing broader questions about scalability, sustainability, and inclusivity. This chapter has focused on the why and how of efficiency—why it is critical to modern machine learning and how to achieve it through a balanced focus on model, compute, and data dimensions. We began by exploring the empirical foundations of scaling laws, revealing how model performance scales with resources and highlighting the critical importance of efficient resource utilization as models grow in complexity. The trade-offs and challenges inherent in scaling, as well as the potential for scaling breakdowns, underscore the necessity of a holistic approach to system design.

By understanding the interdependencies and trade-offs inherent in the algorithmic, compute, and data dimensions of efficiency, we can build systems that align with their operational contexts and long-term objectives. The challenges discussed in this chapter, from the limits of optimization to equity concerns and the tension between efficiency and innovation, highlight the need for a thoughtful approach. Whether working on a high-performance cloud system or a constrained Tiny ML application, the principles of efficiency serve as a compass for navigating the complexities of system design.

The future of scaling laws is a critical area of exploration, particularly as we consider the practical and sustainable limits of continued scaling. Research into the theoretical foundations of scaling, architectural innovations, and the role of data quality will be essential for guiding the development of next-generation AI systems. Moreover, addressing the equity concerns associated with access to compute, data, and efficient models is crucial for ensuring that the benefits of AI are shared broadly.

With this foundation in place, we can now dive into the what, the specific techniques and strategies that enable efficient machine learning systems. By grounding these practices in a clear understanding of the why and the how, we ensure that efficiency remains a guiding principle rather than a reactive afterthought, and that the insights from scaling laws are applied in a way that promotes both performance and sustainability.

9.10 Resources

ℹ Slides

- *Coming soon.*

! Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 10

Model Optimizations

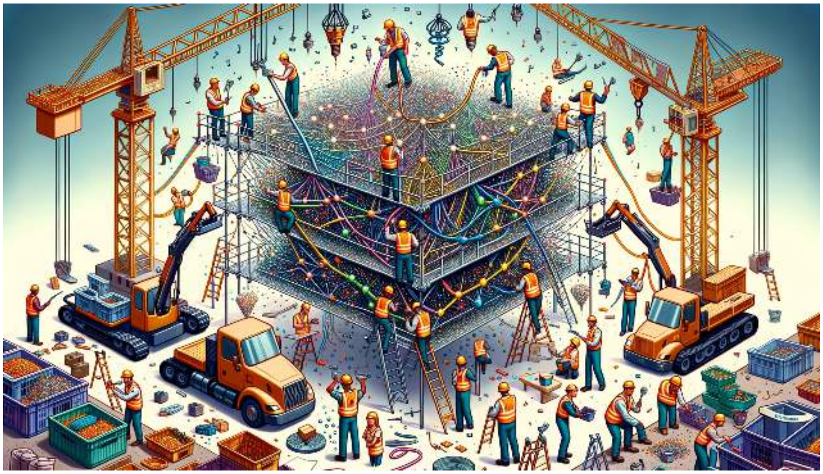


Figure 10.1: *DALL·E 3 Prompt: Illustration of a neural network model represented as a busy construction site, with a diverse group of construction workers, both male and female, of various ethnicities, labeled as 'pruning', 'quantization', and 'sparsity'. They are working together to make the neural network more efficient and smaller, while maintaining high accuracy. The 'pruning' worker, a Hispanic female, is cutting unnecessary connections from the middle of the network. The 'quantization' worker, a Caucasian male, is adjusting or tweaking the weights all over the place. The 'sparsity' worker, an African female, is removing unnecessary nodes to shrink the model. Construction trucks and cranes are in the background, assisting the workers in their tasks. The neural network is visually transforming from a complex and large structure to a more streamlined and smaller one.*

Purpose

How do neural network models transition from design to practical deployment, and what challenges arise in making them efficient and scalable?

Developing machine learning models goes beyond achieving high accuracy; real-world deployment introduces constraints that demand careful adaptation. Models must operate within the limits of computation, memory, latency, and energy efficiency, all while maintaining effectiveness. As models grow in complexity and scale, ensuring their feasibility across diverse hardware and applications becomes increasingly challenging. This necessitates a deeper understanding of the fundamental trade-offs between accuracy and efficiency, as well as the strategies that enable models to function optimally in different environments. By addressing these challenges, we establish guiding principles for transforming machine learning advancements into practical, scalable systems.

Learning Objectives

- Identify, compare, and contrast various techniques for optimizing model representation.
- Assess the trade-offs between different precision reduction strategies.
- Evaluate how hardware-aware model design influences computation and memory efficiency.
- Explain the role of dynamic computation techniques in improving efficiency.
- Analyze the benefits and challenges of sparsity in model optimization and its hardware implications.
- Discuss how different optimization strategies interact and impact system-level performance.

10.1 Overview

As machine learning models evolve in complexity and become increasingly ubiquitous, the focus shifts from solely enhancing accuracy to ensuring that models are practical, scalable, and efficient. The substantial computational requirements for training and deploying state-of-the-art models frequently surpass the limitations imposed by real-world environments, whether in expansive data centers or on resource-constrained mobile devices. Additionally, considerations such as memory constraints, energy consumption, and inference latency critically influence the effective deployment of these models. Model optimization, therefore, serves as the framework that reconciles advanced modeling techniques with practical system limitations, ensuring that enhanced performance is achieved without compromising operational viability.

Definition of Model Optimization

Model Optimization is the *systematic refinement of machine learning models to enhance their efficiency while maintaining effectiveness*. This process involves *balancing trade-offs between accuracy, computational cost, memory usage, latency, and energy efficiency* to ensure models can operate within real-world constraints. Model optimization is driven by fundamental principles such as *reducing redundancy, improving numerical representation, and structuring computations more efficiently*. These principles guide the adaptation of models across *diverse deployment environments*, from cloud-scale infrastructure to resource-constrained edge devices, enabling scalable, practical, and high-performance machine learning systems.

The necessity for model optimization arises from the inherent limitations of modern computational systems. Machine learning models function within a multifaceted ecosystem encompassing hardware capabilities, software frameworks, and diverse deployment scenarios. A model that excels in controlled research environments may prove unsuitable for practical applications due to prohibitive computational costs or substantial memory requirements. Consequently, optimization techniques are critical for aligning high-performing models with the practical constraints of real-world systems.

Optimization is inherently context-dependent. Models deployed in cloud environments often prioritize scalability and throughput, whereas those intended for edge devices must emphasize low power consumption and minimal memory footprint. The array of optimization strategies available enables the adjustment of models to accommodate these divergent constraints without compromising their predictive accuracy.

This chapter explores the principles of model optimization from a systems perspective. Figure 10.2 illustrates the three distinct layers of the optimization stack discussed in the chapter. At the highest level, methodologies aimed at reducing model parameter complexity while preserving inferential capabilities are introduced. Techniques such as pruning and knowledge distillation are examined for their ability to compress and refine models, thereby enhancing model quality and improving system runtime performance.

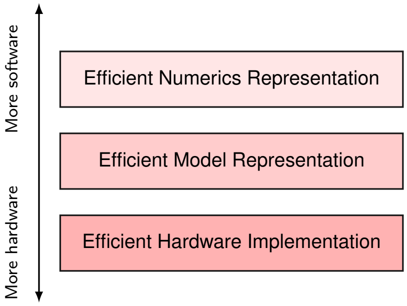

More software
More hardware
Efficient Numerics Representation
Efficient Model Representation
Efficient Hardware Implementation

Figure 10.2: Three layers to be covered.

We also investigate the role of numerical precision in model computations. An understanding of how various numerical representations affect model size, speed, and accuracy is essential for achieving optimal performance. Accordingly, the trade-offs associated with different numerical formats and the implementation of reduced-precision arithmetic are discussed, a topic of particular importance for embedded system deployments where computational resources are constrained.

At the lowest layer, the intricacies of hardware-software co-design are examined, which elucidates how models can be systematically tailored to efficiently utilize the specific characteristics of target hardware platforms. Alignment of machine learning model design with hardware architecture may yield substantial gains in performance and efficiency.

On the whole, the chapter systematically examines the underlying factors that shape optimization approaches, including model representation, numerical precision, and architectural efficiency. In addition, the interdependencies between software and hardware are explored, with emphasis on the roles played by compilers, runtimes, and specialized accelerators in influencing optimization choices. A structured framework is ultimately proposed to guide the selection and application of optimization techniques, ensuring that machine learning models remain both effective and viable under real-world conditions.

10.2 Real-World Models

Machine learning models are rarely deployed in isolation—they operate as part of larger systems with complex constraints, dependencies, and trade-offs. Model optimization, therefore, cannot be treated as a purely algorithmic problem; it must be viewed as a systems-level challenge that considers computational efficiency, scalability, deployment feasibility, and overall system performance. A well-optimized model must balance multiple objectives, including inference speed, memory footprint, power consumption, and accuracy, all while aligning with the specific requirements of the target deployment environment.

Therefore, it is important to understand the systems perspective on model optimization, highlighting why optimization is essential, the key constraints that drive optimization efforts, and the principles that define an effective optimization strategy. By framing optimization as a systems problem, we can move beyond ad-hoc techniques and instead develop principled approaches that integrate hardware, software, and algorithmic considerations into a unified optimization framework.

10.2.1 Practical Models

Modern machine learning models often achieve impressive accuracy on benchmark datasets, but making them practical for real-world use is far from trivial. In practice, machine learning systems operate under a range of computational, memory, latency, and energy constraints that significantly impact both training and inference (Choudhary et al. 2020). A model that performs well in a research setting may be impractical when integrated into a broader system, whether it is deployed in the cloud, embedded in a smartphone, or running on a tiny microcontroller.

The real-world feasibility of a model depends on more than just accuracy—it also hinges on how efficiently it can be trained, stored, and executed. In large-scale Cloud ML settings, optimizing models helps minimize training time, computational cost, and power consumption, making large-scale AI workloads more efficient (Jeff Dean, Patterson, and Young 2018). In contrast, Edge ML requires models to run with limited compute resources, necessitating optimizations that reduce memory footprint and computational complexity. Mobile ML introduces additional constraints, such as battery life and real-time responsiveness, while Tiny ML pushes efficiency to the extreme, requiring models to fit within the memory and processing limits of ultra-low-power devices (C. R. Banbury et al. 2020).

Optimization also plays a crucial role in making AI more sustainable and accessible. Reducing a model's energy footprint is critical as AI workloads scale, helping mitigate the environmental impact of large-scale ML training and inference (D. Patterson et al. 2021b). At the same time, optimized models can expand the reach of machine learning, enabling applications in low-resource environments, from rural healthcare to autonomous systems operating in the field.

Ultimately, without systematic optimization, many machine learning models remain confined to academic studies rather than progressing to practical applications. For ML systems engineers and practitioners, the primary objective is to bridge the gap between theoretical potential and real-world functionality by deliberately designing models that are both efficient in execution and robust in diverse operational environments.

10.2.2 Accuracy-Efficiency Balance

Machine learning models are typically optimized to achieve high accuracy, but improving accuracy often comes at the cost of increased computational complexity. Larger models with more parameters, deeper architectures, and higher numerical precision can yield better performance on benchmark tasks. However, these improvements introduce challenges related to memory footprint, inference latency, power consumption, and training efficiency. As machine learning systems are deployed across a wide range of hardware platforms, balancing accuracy and efficiency becomes a fundamental challenge in model optimization.

From a systems perspective, accuracy and efficiency are often in direct tension. Increasing model capacity[0], including more parameters, deeper layers, and larger input resolutions, generally enhances predictive performance. However, these same modifications also increase computational cost, making inference slower and more resource-intensive. Similarly, during training, larger models demand greater memory bandwidth, longer training times, and more energy consumption, all of which introduce scalability concerns.

The need for efficiency constraints extends beyond inference. Training efficiency is critical for both research and industrial-scale applications, as larger models require greater computational resources and longer convergence times. Unoptimized training pipelines can result in prohibitive costs and delays, limiting the pace of innovation and deployment. On the inference side, real-time applications impose strict constraints on latency and power consumption, further motivating the need for optimization.

Balancing accuracy and efficiency requires a structured approach to model optimization, where trade-offs are carefully analyzed rather than applied indiscriminately. Some optimizations, such as pruning[1] redundant parameters or reducing numerical precision, can improve efficiency without significantly impacting accuracy. Other techniques, like model distillation[2] or architecture search, aim to preserve predictive performance while improving computational efficiency. The key challenge is to systematically determine which optimizations provide the best trade-offs for a given application and hardware platform.

10.2.3 Optimization System Constraints

Machine learning models operate within a set of fundamental system constraints that influence how they are designed, trained, and deployed. These constraints arise from the computational resources available, the hardware on

[0] Model Capacity: The ability of a model to capture and represent the complexity of a dataset, influenced by parameters and architecture.

[1] Pruning: The process of removing unnecessary parameters from a neural network to reduce its size and improve efficiency.

[2] Model Distillation: A technique to compress a large model into a smaller, more efficient model while retaining performance.

which the model runs, and the operational requirements of the application. Understanding these constraints is essential for developing effective optimization strategies that balance accuracy, efficiency, and feasibility. The primary system constraints that drive model optimization include:

Computational Cost: Training and inference require significant compute resources, especially for large-scale models. The computational complexity of a model affects the feasibility of training on large datasets and deploying real-time inference workloads. Optimization techniques that reduce computation, including pruning, quantization, and efficient architectures, can significantly lower costs.

Memory and Storage Limitations: Models must fit within the memory constraints of the target system. This includes RAM[3] limitations during execution and storage constraints for model persistence. Large models with billions of parameters may exceed the capacity of edge devices or embedded systems, necessitating optimizations that reduce memory footprint without compromising performance.

Latency and Throughput: Many applications impose real-time constraints, requiring models to produce predictions within strict latency budgets. In autonomous systems, healthcare diagnostics, and interactive AI applications, slow inference times can render a model unusable. Optimizing model execution, by employing reduced precision arithmetic, optimizing data movement, or utilizing parallel computation, can help meet real-time constraints.

Energy Efficiency and Power Consumption: Power constraints are critical in mobile, edge, and embedded AI systems. High energy consumption impacts battery-powered devices and increases operational costs in large-scale cloud deployments. Techniques such as model sparsity,[4] adaptive computation,[5] and hardware-aware optimization contribute to energy-efficient AI.

Scalability and Hardware Compatibility: Model optimizations must align with the capabilities of the target hardware. A model optimized for specialized accelerators (e.g., GPUs, TPUs, FPGAs) may not perform efficiently on general-purpose CPUs. Additionally, scaling models across distributed systems introduces new challenges in synchronization and workload balancing.

These constraints are interdependent, meaning that optimizing for one factor may impact another. For example, reducing numerical precision can lower memory usage and improve inference speed but may introduce quantization errors that degrade accuracy. Similarly, aggressive pruning can reduce computation but may lead to diminished generalization if not carefully managed.

10.3 Model Optimization Dimensions

Machine learning models must balance accuracy, efficiency, and feasibility to operate effectively in real-world systems. As discussed in the previous section, optimization is necessary to address key system constraints such as computational cost, memory limitations, energy efficiency, and latency requirements. However, model optimization is not a single technique but a structured process that can be categorized into three fundamental dimensions: model representation optimization, numerical precision optimization, and architectural efficiency optimization.

[3] Random Access Memory (RAM): Hardware feature that provides fast, volatile working memory for temporary data storage during program execution. Unlike persistent storage devices like the hard disk, RAM enables rapid data access but loses its contents when powered off, making it critical for efficient computation and memory-intensive operations.

[4] Model Sparsity: Refers to techniques that involve using fewer non-zero parameters within a model to reduce complexity and increase speed.

[5] Adaptive Computation: Dynamic adjustment of computational resources based on the task complexity to optimize efficiency.

Each of these dimensions addresses a distinct aspect of efficiency. Model representation optimization focuses on modifying the architecture of the model itself to reduce redundancy while preserving accuracy. Numerical precision optimization improves efficiency by adjusting how numerical values are stored and computed, reducing the computational and memory overhead of machine learning operations. Architectural efficiency focuses on optimizing how computations are executed, ensuring that operations are performed efficiently across different hardware platforms.

Understanding these three dimensions provides a structured framework for systematically improving model efficiency. Rather than applying ad hoc techniques, machine learning practitioners must carefully select optimizations based on their impact across these dimensions, considering trade-offs between accuracy, efficiency, and deployment constraints.

10.3.1 Model Representation

The first dimension, model representation optimization, focuses on reducing redundancy in the structure of machine learning models. Large models often contain excessive parameters that contribute little to overall performance but significantly increase memory footprint and computational cost. Optimizing model representation involves techniques that remove unnecessary components while maintaining predictive accuracy. Common approaches include pruning, which eliminates redundant weights and neurons, and knowledge distillation, where a smaller model learns to approximate the behavior of a larger model. Additionally, automated architecture search methods refine model structures to balance efficiency and accuracy. These optimizations primarily impact how models are designed at an algorithmic level, ensuring that they remain effective while being computationally manageable.

10.3.2 Numerical Precision

The second dimension, numerical precision optimization, addresses how numerical values are represented and processed within machine learning models. Reducing the precision of computations can significantly lower the memory and computational requirements of a model, particularly for machine learning workloads. Quantization techniques map high-precision weights and activations to lower-bit representations, enabling efficient execution on hardware accelerators such as GPUs, TPUs, and specialized AI chips. Mixed-precision training dynamically adjusts precision levels during training to strike a balance between efficiency and accuracy. By carefully optimizing numerical precision, models can achieve substantial reductions in computational cost while maintaining acceptable levels of accuracy.

10.3.3 Architectural Efficiency

The third dimension, architectural efficiency, focuses on how computations are performed efficiently during both training and inference. A well-designed model structure is not sufficient if its execution is suboptimal. Many machine learning models contain redundancies in their computational graphs, leading to inefficiencies in how operations are scheduled and executed. Architectural efficiency involves techniques that exploit sparsity in both model weights and activations, factorize large computational components into more efficient forms, and dynamically adjust computation based on input complexity. These methods improve execution efficiency across different hardware platforms, reducing latency and power consumption. In addition to inference optimizations, architectural efficiency also applies to training, where techniques such as gradient checkpointing and low-rank adaptation help reduce memory overhead and computational demands.

10.3.4 Tripartite Framework

These three dimensions collectively provide a framework for understanding model optimization. While each category targets different aspects of efficiency, they are highly interconnected. Pruning, for example, primarily falls under model representation but also affects architectural efficiency by reducing the number of operations performed during inference. Quantization reduces numerical precision but can also impact memory footprint and execution efficiency. Understanding these interdependencies is crucial for selecting the right combination of optimizations for a given system.

The choice of optimizations is driven by system constraints, which define the practical limitations within which models must operate. A machine learning model deployed in a data center has different constraints from one running on a mobile device or an embedded system. Computational cost, memory usage, inference latency, and energy efficiency all influence which optimizations are most appropriate for a given scenario. A model that is too large for a resource-constrained device may require aggressive pruning and quantization, while a latency-sensitive application may benefit from operator fusion[6] and hardware-aware scheduling[7].

Table 10.1 summarizes how different system constraints map to the three core dimensions of model optimization.

Table 10.1: Mapping of system constraints to optimization dimensions.

System Constraint	Model Representation	Numerical Precision	Architectural Efficiency
Computational Cost		✓	✓
Memory and Storage	✓	✓	
Latency and Throughput	✓		✓
Energy Efficiency		✓	✓
Scalability	✓		✓

This mapping highlights the interdependence between optimization strategies and real-world constraints. Although each system constraint primarily aligns with one or more optimization dimensions, the relationships are not strictly one-to-one. Many optimization techniques affect multiple constraints simultaneously. Structuring model optimization along these three dimensions and mapping techniques to specific system constraints enables practitioners to analyze trade-offs more effectively and select optimizations that best align with

[6] Operator Fusion: A technique that merges multiple operations into a single operation to reduce computational overhead.

[7] Hardware-Aware Scheduling: Optimizing computational tasks based on the specific hardware characteristics.

deployment requirements. The following sections explore each optimization dimension in detail, highlighting the key techniques and their impact on model efficiency.

10.4 Model Representation Optimization

Model representation plays a key role in determining the computational and memory efficiency of a machine learning system. The way a model is structured, not just in terms of the number of parameters but also how these parameters interact, directly affects its ability to scale, deploy efficiently, and generalize effectively. Optimizing model representation involves reducing redundancy, restructuring architectures for efficiency, and leveraging automated design methods to find optimal configurations.

The primary goal of model representation optimization is to eliminate unnecessary complexity while preserving model performance. Many state-of-the-art models are designed to maximize accuracy with little regard for efficiency, leading to architectures with excessive parameters, redundant computations, and inefficient data flow. In real-world deployment scenarios, these inefficiencies translate into higher computational costs, increased memory usage, and slower inference times. Addressing these issues requires systematically restructuring the model to remove redundancy, minimize unnecessary computations, and ensure that every parameter contributes meaningfully to the task at hand.

From a systems perspective, model representation optimization focuses on two key objectives. First, reducing redundancy by eliminating unnecessary parameters, neurons, or layers while preserving model accuracy. Many models are overparameterized, meaning that a smaller version could achieve similar performance with significantly lower computational overhead. Second, structuring computations efficiently to ensure that the model's architecture aligns well with modern hardware capabilities, such as leveraging parallel processing and minimizing costly memory operations. An unoptimized model may be unnecessarily large, leading to slower inference times, higher energy consumption, and increased deployment costs. Conversely, an overly compressed model may lose too much predictive accuracy, making it unreliable for real-world use. The challenge in model representation optimization is to strike a balance between model size, accuracy, and efficiency, selecting techniques that reduce computational complexity while maintaining strong generalization.

To systematically approach model representation optimization, we focus on three key techniques that have proven effective in balancing efficiency and accuracy. Pruning systematically removes parameters or entire structural components that contribute little to overall performance, reducing computational and memory overhead while preserving accuracy. Knowledge distillation transfers knowledge from a large, high-capacity model to a smaller, more efficient model, enabling smaller models to retain predictive power while reducing computational cost. Finally, NAS automates the process of designing models optimized for specific constraints, leveraging machine learning itself to explore and refine model architectures.

We focus on these three techniques because they represent distinct but complementary approaches to optimizing model representation. Pruning and knowledge distillation focus on reducing redundancy in existing models, while NAS addresses how to build optimized architectures from the ground up. Together, they provide a structured framework for understanding how to create machine learning models that are both accurate and computationally efficient. Each of these techniques offers a different approach to improving model efficiency, and in many cases, they can be combined to achieve even greater optimization.

10.4.1 Pruning

State-of-the-art machine learning models often contain millions, or even billions, of parameters, many of which contribute minimally to final predictions. While large models enhance representational power and generalization, they also introduce inefficiencies that impact both training and deployment. From a machine learning systems perspective, these inefficiencies present several challenges:

1. **High Memory Requirements**: Large models require substantial storage, limiting their feasibility on resource-constrained devices such as smartphones, IoT devices, and embedded systems. Storing and loading these models also creates bandwidth bottlenecks in distributed ML pipelines.
2. **Increased Computational Cost**: More parameters lead to higher inference latency and energy consumption, which is particularly problematic for real-time applications such as autonomous systems, speech recognition, and mobile AI. Running unoptimized models on hardware accelerators like GPUs and TPUs requires additional compute cycles, increasing operational costs.
3. **Scalability Limitations**: Training and deploying large models at scale is resource-intensive in terms of compute, memory, and power. Large-scale distributed training demands high-bandwidth communication and storage, while inference in production environments becomes costly without optimizations.

Despite these challenges, not all parameters in a model are necessary to maintain accuracy. Many weights contribute little to the decision-making process, and their removal can significantly improve efficiency without substantial performance degradation. This motivates the use of pruning, a class of optimization techniques that systematically remove redundant parameters while preserving model accuracy.

ℹ Definition of Pruning

Pruning is a model optimization technique that removes unnecessary parameters from a neural network while maintaining predictive performance. By systematically eliminating redundant weights, neurons, or layers, pruning reduces model size and computational cost, making it more efficient for storage, inference, and deployment.

Pruning allows models to become smaller, faster, and more efficient without requiring fundamental changes to their architecture. By reducing redundancy, pruning directly addresses the memory, computation, and scalability constraints of machine learning systems, making it a key optimization technique for deploying ML models across cloud, edge, and mobile platforms.

10.4.1.1 Distillation Mathematics

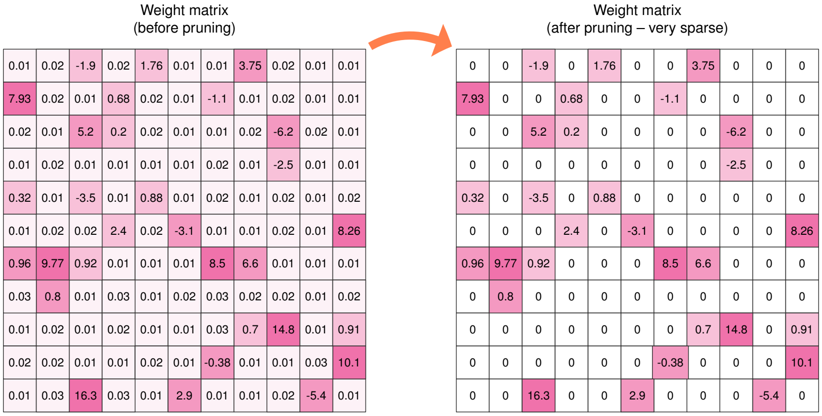
Pruning can be formally described as an optimization problem, where the goal is to reduce the number of parameters in a neural network while maintaining its predictive performance. Given a trained model with parameters W, pruning seeks to find a sparse version of the model, $\hat{W}$, that retains only the most important parameters. The objective can be expressed as:

$$\min_{\hat{W}} \mathcal{L}(\hat{W}) \quad \text{subject to} \quad \|\hat{W}\|_0 \leq k$$

where:

- $\mathcal{L}(\hat{W})$ represents the model's loss function after pruning.
- $\hat{W}$ denotes the pruned model's parameters.
- $\|\hat{W}\|_0$ is the number of nonzero parameters in $\hat{W}$, constrained to a budget k.

As illustrated in Figure 10.3, pruning reduces the number of nonzero weights by eliminating small-magnitude values, transforming a dense weight matrix into a sparse representation. This explicit enforcement of sparsity aligns with the ℓ_0-norm constraint in our optimization formulation.



0.01	0.02	-1.9	0.02	1.76	0.01	0.01	3.75	0.02	0.01	0.01
7.93	0.02	0.01	0.68	0.02	0.01	-1.1	0.01	0.02	0.01	0.01
0.02	0.01	5.2	0.2	0.02	0.01	0.01	0.02	-6.2	0.02	0.01
0.01	0.02	0.01	0.01	0.01	0.01	0.02	0.01	-2.5	0.01	0.01
0.32	0.01	-3.5	0.01	0.88	0.01	0.02	0.01	0.02	0.01	0.02
0.01	0.02	0.02	2.4	0.02	-3.1	0.01	0.01	0.02	0.01	8.26
0.96	9.77	0.92	0.01	0.01	0.01	8.5	6.6	0.01	0.01	0.01
0.03	0.8	0.01	0.03	0.01	0.02	0.03	0.02	0.02	0.01	0.02
0.01	0.02	0.01	0.02	0.01	0.01	0.03	0.7	14.8	0.01	0.91
0.02	0.02	0.01	0.01	0.02	0.01	-0.38	0.01	0.01	0.03	10.1
0.01	0.03	16.3	0.03	0.01	2.9	0.01	0.01	0.02	-5.4	0.01

0	0	-1.9	0	1.76	0	0	3.75	0	0	0
7.93	0	0	0.68	0	0	-1.1	0	0	0	0
0	0	5.2	0.2	0	0	0	0	-6.2	0	0
0	0	0	0	0	0	0	0	-2.5	0	0
0.32	0	-3.5	0	0.88	0	0	0	0	0	0
0	0	0	2.4	0	-3.1	0	0	0	0	8.26
0.96	9.77	0.92	0	0	0	8.5	6.6	0	0	0
0	0.8	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0.7	14.8	0	0.91
0	0	0	0	0	0	-0.38	0	0	0	10.1
0	0	16.3	0	0	2.9	0	0	0	-5.4	0

Figure 10.3: Weight matrix before and after pruning.

However, solving this problem exactly is computationally infeasible due to the discrete nature of the ℓ_0-norm constraint. Finding the optimal subset of parameters to retain would require evaluating an exponential number of possible parameter configurations, making it impractical for deep networks with millions of parameters (Labarge, n.d.).

To make pruning computationally feasible, practical methods replace the hard constraint on the number of remaining parameters with a soft regularization term that encourages sparsity. A common relaxation is to introduce an ℓ_1-norm regularization penalty, leading to the following objective:

$$\min_{W} \mathcal{L}(W) + \lambda \|W\|_1$$

where λ controls the degree of sparsity. The ℓ_1-norm encourages smaller weight values and promotes sparsity but does not strictly enforce zero values. Other methods use iterative heuristics, where parameters with the smallest magnitudes are pruned in successive steps, followed by fine-tuning to recover lost accuracy (Gale, Elsen, and Hooker 2019a).

10.4.1.2 Target Structures

Pruning methods vary based on which structures within a neural network are removed. The primary targets include neurons, channels, and layers, each with distinct implications for the model's architecture and performance.

- **Neuron pruning** removes entire neurons along with their associated weights and biases, reducing the width of a layer. This technique is often applied to fully connected layers.
- **Channel pruning** (or filter pruning), commonly used in convolutional neural networks, eliminates entire channels or filters. This reduces the depth of feature maps, which impacts the network's ability to extract certain features. Channel pruning is particularly valuable in image-processing tasks where computational efficiency is a priority.
- **Layer pruning** removes entire layers from the network, significantly reducing depth. While this approach can yield substantial efficiency gains, it requires careful balance to ensure the model retains sufficient capacity to capture complex patterns.

Figure 10.4 illustrates the differences between channel pruning and layer pruning. When a channel is pruned, the model's architecture must be adjusted to accommodate the structural change. Specifically, the number of input channels in subsequent layers must be modified, requiring alterations to the depths of the filters applied to the layer with the removed channel. In contrast, layer pruning removes all channels within a layer, necessitating more substantial architectural modifications. In this case, connections between remaining layers must be reconfigured to bypass the removed layer. Regardless of the pruning approach, fine-tuning is essential to adapt the remaining network and restore performance.

10.4.1.3 Unstructured Pruning

Unstructured pruning reduces the number of active parameters in a neural network by removing individual weights while preserving the overall network architecture. Many machine learning models are overparameterized, meaning they contain more weights than are strictly necessary for accurate predictions. During training, some connections become redundant, contributing little to

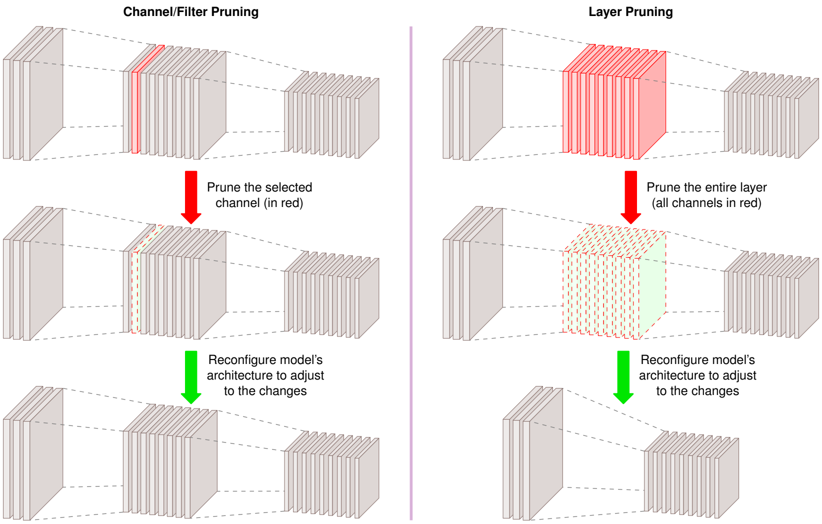

Channel/Filter Pruning
Layer Pruning
Prune the selected channel (in red)
Prune the entire layer (all channels in red)
Reconfigure model's architecture to adjust to the changes
Reconfigure model's architecture to adjust to the changes

Figure 10.4: Channel vs layer pruning.

the final computation. Pruning these weak connections can reduce memory requirements while preserving most of the model's accuracy.

Mathematically, unstructured pruning introduces sparsity into the weight matrices of a neural network. Let $W \in \mathbb{R}^{m \times n}$ represent a weight matrix in a given layer of a network. Pruning removes a subset of weights by applying a binary mask $M \in \{0,1\}^{m \times n}$, yielding a pruned weight matrix:

$$\hat{W} = M \odot W$$

where $\odot$ represents the element-wise Hadamard product[8]. The mask M is constructed based on a pruning criterion, typically weight magnitude. A common approach is magnitude-based pruning, which removes a fraction s of the lowest-magnitude weights. This is achieved by defining a threshold τ such that:

$$M_{i,j} = \begin{cases} 1, & \text{if } |W_{i,j}| > \tau \\ 0, & \text{otherwise} \end{cases}$$

where τ is chosen to ensure that only the largest $(1-s)$ fraction of weights remain. This method assumes that larger-magnitude weights contribute more to the network's function, making them preferable for retention.

The primary advantage of unstructured pruning is memory efficiency. By reducing the number of nonzero parameters, pruned models require less storage, which is particularly beneficial when deploying models to embedded or mobile devices with limited memory.

However, unstructured pruning does not necessarily improve computational efficiency on modern machine learning hardware. Standard GPUs and TPUs are optimized for dense matrix multiplications, and a sparse weight matrix often cannot fully utilize hardware acceleration unless specialized sparse computation kernels are available. Consequently, unstructured pruning is most

[8] Hadamard Product: An element-wise product operation between two matrices of the same dimensions.

beneficial when the goal is to compress a model for storage rather than to accelerate inference speed. While unstructured pruning improves model efficiency at the parameter level, it does not alter the structural organization of the network.

10.4.1.4 Structured Pruning

While unstructured pruning removes individual weights from a neural network, structured pruning eliminates entire computational units, such as neurons, filters, channels, or layers. This approach is particularly beneficial for hardware efficiency, as it produces smaller dense models that can be directly mapped to modern machine learning accelerators. Unlike unstructured pruning, which results in sparse weight matrices that require specialized execution kernels to exploit computational benefits, structured pruning leads to more efficient inference on general-purpose hardware by reducing the overall size of the network architecture.

Structured pruning is motivated by the observation that not all neurons, filters, or layers contribute equally to a model's predictions. Some units primarily carry redundant or low-impact information, and removing them does not significantly degrade model performance. The challenge lies in identifying which structures can be pruned while preserving accuracy.

Figure 10.5 illustrates the key differences between unstructured and structured pruning. On the left, unstructured pruning removes individual weights (depicted as dashed connections), leading to a sparse weight matrix. This can disrupt the original structure of the network, as shown in the fully connected network where certain connections have been randomly pruned. While this can reduce the number of active parameters, the resulting sparsity requires specialized execution kernels to fully leverage computational benefits.

In contrast, structured pruning, depicted in the middle and right sections of the figure, removes entire neurons (dashed circles) or filters while preserving the network's overall structure. In the middle section, a pruned fully connected network retains its fully connected nature but with fewer neurons. On the right, structured pruning is applied to a convolutional neural network (CNN) by removing convolutional kernels or entire channels (dashed squares). This method maintains the CNN's fundamental convolutional operations while reducing the computational load, making it more compatible with hardware accelerators.

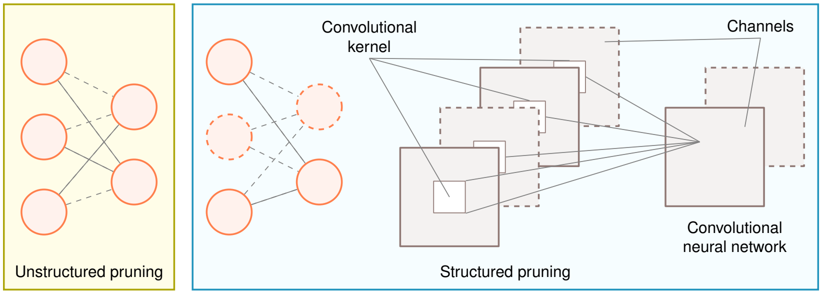


Figure 10.5: Unstructured vs structured pruning. Source: C. Qi et al. (2021).

A common approach to structured pruning is magnitude-based pruning, where entire neurons or filters are removed based on the magnitude of their associated weights. The intuition behind this method is that parameters with smaller magnitudes contribute less to the model's output, making them prime candidates for elimination. The importance of a neuron or filter is often measured using a norm function, such as the ℓ_1-norm or ℓ_2-norm, applied to the weights associated with that unit. If the norm falls below a predefined threshold, the corresponding neuron or filter is pruned. This method is straightforward to implement and does not require additional computational overhead beyond computing norms across layers.

Another strategy is activation-based pruning, which evaluates the average activation values of neurons or filters over a dataset. Neurons that consistently produce low activations contribute less information to the network's decision process and can be safely removed. This method captures the dynamic behavior of the network rather than relying solely on static weight values. Activation-based pruning requires profiling the model over a representative dataset to estimate the average activation magnitudes before making pruning decisions.

Gradient-based pruning leverages information from the model's training process to identify less significant neurons or filters. The key idea is that units with smaller gradient magnitudes contribute less to reducing the loss function, making them less critical for learning. By ranking neurons based on their gradient values, structured pruning can remove those with the least impact on model optimization. Unlike magnitude-based or activation-based pruning, which rely on static properties of the trained model, gradient-based pruning requires access to gradient computations and is typically applied during training rather than as a post-processing step.

Each of these methods presents trade-offs in terms of computational complexity and effectiveness. Magnitude-based pruning is computationally inexpensive and easy to implement but does not account for how neurons behave across different data distributions. Activation-based pruning provides a more data-driven pruning approach but requires additional computations to estimate neuron importance. Gradient-based pruning leverages training dynamics but may introduce additional complexity if applied to large-scale models. The choice of method depends on the specific constraints of the target deployment environment and the performance requirements of the pruned model.

10.4.1.5 Dynamic Pruning

Traditional pruning methods, whether unstructured or structured, typically involve static pruning, where parameters are permanently removed after training or at fixed intervals during training. However, this approach assumes that the importance of parameters is fixed, which is not always the case. In contrast, dynamic pruning adapts pruning decisions based on the input data or training dynamics, allowing the model to adjust its structure in real time.

Dynamic pruning can be implemented using runtime sparsity techniques, where the model actively determines which parameters to utilize based on input characteristics. Activation-conditioned pruning exemplifies this approach by selectively deactivating neurons or channels that exhibit low activation values for specific inputs (J. Hu et al. 2023). This method introduces input-dependent sparsity patterns, effectively reducing the computational workload during inference without permanently modifying the model architecture.

For instance, consider a convolutional neural network processing images with varying complexity. During inference of a simple image containing mostly uniform regions, many convolutional filters may produce negligible activations. Dynamic pruning identifies these low-impact filters and temporarily excludes them from computation, improving efficiency while maintaining accuracy for the current input. This adaptive behavior is particularly advantageous in latency-sensitive applications, where computational resources must be allocated judiciously based on input complexity.

Another class of dynamic pruning operates during training, where sparsity is gradually introduced and adjusted throughout the optimization process. Methods such as gradual magnitude pruning start with a dense network and progressively increase the fraction of pruned parameters as training progresses. Instead of permanently removing parameters, these approaches allow the network to recover from pruning-induced capacity loss by regrowing connections that prove to be important in later stages of training.

Dynamic pruning presents several advantages over static pruning. It allows models to adapt to different workloads, potentially improving efficiency while maintaining accuracy. Unlike static pruning, which risks over-pruning and degrading performance, dynamic pruning provides a mechanism for selectively reactivating parameters when necessary. However, implementing dynamic pruning requires additional computational overhead, as pruning decisions must be made in real-time, either during training or inference. This makes it more complex to integrate into standard machine learning pipelines compared to static pruning.

Despite its challenges, dynamic pruning is particularly useful in edge computing and adaptive AI systems, where resource constraints and real-time efficiency requirements vary across different inputs. The next section explores the practical considerations and trade-offs involved in choosing the right pruning method for a given machine learning system.

10.4.1.6 Pruning Trade-offs

Pruning techniques offer different trade-offs in terms of memory efficiency, computational efficiency, accuracy retention, hardware compatibility, and implementation complexity. The choice of pruning strategy depends on the specific constraints of the machine learning system and the deployment environment.

Unstructured pruning is particularly effective in reducing model size and memory footprint, as it removes individual weights while keeping the overall model architecture intact. However, since machine learning accelerators are optimized for dense matrix operations, unstructured pruning does not always translate to significant computational speed-ups unless specialized sparse execution kernels are used.

Structured pruning, in contrast, eliminates entire neurons, channels, or layers, leading to a more hardware-friendly model. This technique provides direct computational savings, as it reduces the number of floating-point operations

(FLOPs) required during inference. The downside is that modifying the network structure can lead to a greater accuracy drop, requiring careful fine-tuning to recover lost performance.

Dynamic pruning introduces adaptability into the pruning process by adjusting which parameters are pruned at runtime based on input data or training dynamics. This allows for a better balance between accuracy and efficiency, as the model retains the flexibility to reintroduce previously pruned parameters if needed. However, dynamic pruning increases implementation complexity, as it requires additional computations to determine which parameters to prune on-the-fly.

Table 10.2 summarizes the key structural differences between these pruning approaches, outlining how each method modifies the model and impacts its execution.

Table 10.2: Comparison of unstructured, structured, and dynamic pruning.

Aspect	Unstructured Pruning	Structured Pruning	Dynamic Pruning
What is removed?	Individual weights in the model	Entire neurons, channels, filters, or layers	Adjusts pruning based on runtime conditions
Model structure	Sparse weight matrices; original architecture remains unchanged	Model architecture is modified; pruned layers are fully removed	Structure adapts dynamically
Impact on memory	Reduces model storage by eliminating nonzero weights	Reduces model storage by removing entire components	Varies based on real-time pruning
Impact on computation	Limited; dense matrix operations still required unless specialized sparse computation is used	Directly reduces FLOPs and speeds up inference	Balances accuracy and efficiency dynamically
Hardware compatibility	Sparse weight matrices require specialized execution support for efficiency	Works efficiently with standard deep learning hardware	Requires adaptive inference engines
Fine-tuning required?	Often necessary to recover accuracy after pruning	More likely to require fine-tuning due to larger structural modifications	Adjusts dynamically, reducing the need for retraining
Use cases	Memory-efficient model compression, particularly for cloud deployment	Real-time inference optimization, mobile/edge AI, and efficient training	Adaptive AI applications, real-time systems

10.4.1.7 Pruning Strategies

Beyond the broad categories of unstructured, structured, and dynamic pruning, different pruning workflows can impact model efficiency and accuracy retention. Two widely used pruning strategies are iterative pruning and one-shot pruning, each with its own benefits and trade-offs.

Iterative Pruning. Iterative pruning implements a gradual approach to structure removal through multiple cycles of pruning followed by fine-tuning. During each cycle, the algorithm removes a small subset of structures based on predefined importance metrics. The model then undergoes fine-tuning to adapt to these structural modifications before proceeding to the next pruning iteration. This methodical approach helps prevent sudden drops in accuracy while allowing the network to progressively adjust to reduced complexity.

To illustrate this process, consider pruning six channels from a convolutional neural network as shown in Figure 10.6. Rather than removing all channels simultaneously, iterative pruning eliminates two channels per iteration over three cycles. Following each pruning step, the model undergoes fine-tuning to recover performance. The first iteration, which removes two channels, results in an accuracy decrease from 0.995 to 0.971, but subsequent fine-tuning restores accuracy to 0.992. After completing two additional pruning-tuning cycles, the final model achieves 0.991 accuracy, which represents only a 0.4% reduction from the original, while operating with 27% fewer channels. By distributing structural modifications across multiple iterations, the network maintains its performance capabilities while achieving improved computational efficiency.

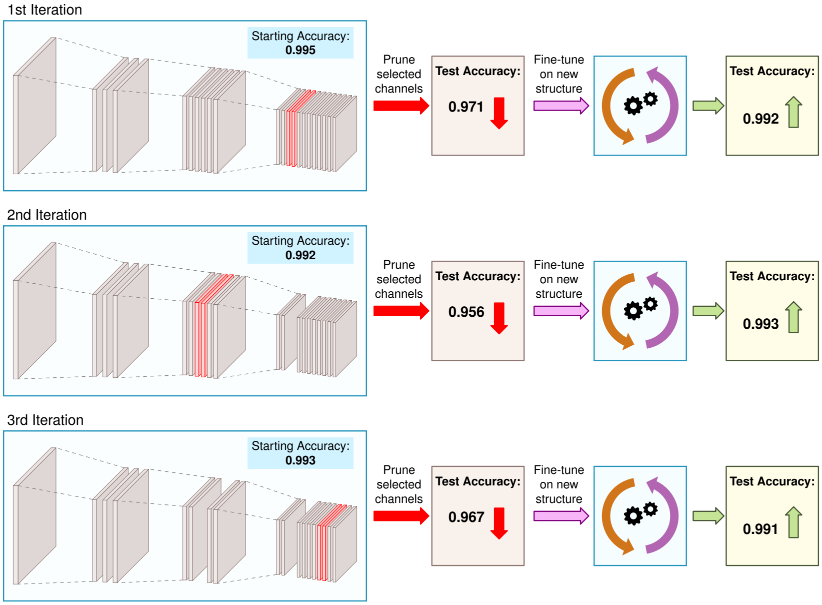


Figure 10.6: Iterative pruning.

One-shot Pruning. One-shot pruning removes multiple architectural components in a single step, followed by an extensive fine-tuning phase to recover model accuracy. This aggressive approach compresses the model quickly but risks greater accuracy degradation, as the network must adapt to substantial structural changes simultaneously.

Consider applying one-shot pruning to the same network discussed in the iterative pruning example. Instead of removing two channels at a time over multiple iterations, one-shot pruning eliminates all six channels at once, as illustrated in Figure 10.7. Removing 27% of the network's channels simultaneously causes the accuracy to drop significantly, from 0.995 to 0.914. Even after fine-tuning, the network only recovers to an accuracy of 0.943, which is a 5% degradation from the original unpruned network. While both iterative and one-shot pruning ultimately produce identical network structures, the gradual approach of iterative pruning better preserves model performance.

The choice of pruning strategy requires careful consideration of several key factors that influence both model efficiency and performance:

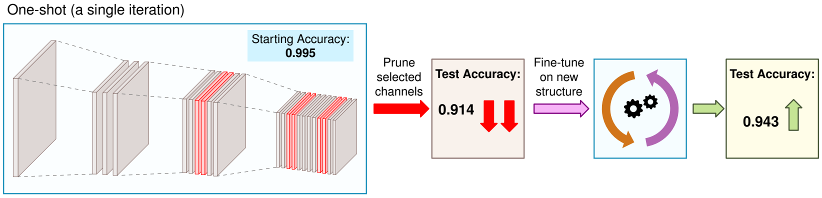


Figure 10.7: One-shot pruning.

Sparsity Target: The desired level of parameter reduction directly impacts strategy selection. Higher reduction targets often necessitate iterative approaches to maintain accuracy, while moderate sparsity goals may be achievable through simpler one-shot methods.

Computational Resources: Available computing power significantly influences strategy choice. Iterative pruning demands substantial resources for multiple fine-tuning cycles, whereas one-shot approaches require fewer resources but may sacrifice accuracy.

Performance Requirements: Applications with strict accuracy requirements typically benefit from gradual, iterative pruning to carefully preserve model capabilities. Use cases with more flexible performance constraints may accommodate more aggressive one-shot approaches.

Development Timeline: Project schedules impact pruning decisions. One-shot methods enable faster deployment when time is limited, though iterative approaches generally achieve superior results given sufficient optimization periods.

Hardware Constraints: Target platform capabilities significantly influence strategy selection. Certain hardware architectures may better support specific sparsity patterns, making particular pruning approaches more advantageous for deployment.

The choice between pruning strategies requires careful evaluation of project requirements and constraints. One-shot pruning enables rapid model compression by removing multiple parameters simultaneously, making it suitable for scenarios where deployment speed is prioritized over accuracy. However, this aggressive approach often results in greater performance degradation compared to more gradual methods. Iterative pruning, on the other hand, while computationally intensive and time-consuming, typically achieves superior accuracy retention through systematic parameter reduction across multiple cycles. This methodical approach enables the network to adapt progressively to structural modifications, preserving critical connections that maintain model performance. The trade-off is increased optimization time and computational overhead. By evaluating these factors systematically, practitioners can select a pruning approach that optimally balances efficiency gains with model performance for their specific use case.

10.4.1.8 Lottery Ticket Hypothesis

Pruning is widely used to reduce the size and computational cost of neural networks, but the process of determining which parameters to remove is not always straightforward. While traditional pruning methods eliminate weights based on magnitude, structure, or dynamic conditions, recent research suggests that pruning is not just about reducing redundancy; it may also reveal inherently efficient subnetworks that exist within the original model.

This perspective leads to the Lottery Ticket Hypothesis (LTH), which challenges conventional pruning workflows by proposing that within large neural networks, there exist small, well-initialized subnetworks, referred to as ‘winning tickets’ (see footnote for details), that can achieve comparable accuracy to the full model when trained in isolation. Rather than viewing pruning as just a post-training compression step, LTH suggests it can serve as a discovery mechanism to identify these efficient subnetworks early in training.

LTH is validated through an iterative pruning process, illustrated in Figure 10.8. A large network is first trained to convergence. The lowest-magnitude weights are then pruned, and the remaining weights are reset to their original initialization rather than being re-randomized. This process is repeated iteratively, gradually reducing the network’s size while preserving performance. After multiple iterations, the remaining subnetwork, referred to as the ‘winning ticket’[9], proves capable of training to the same or higher accuracy as the original full model.

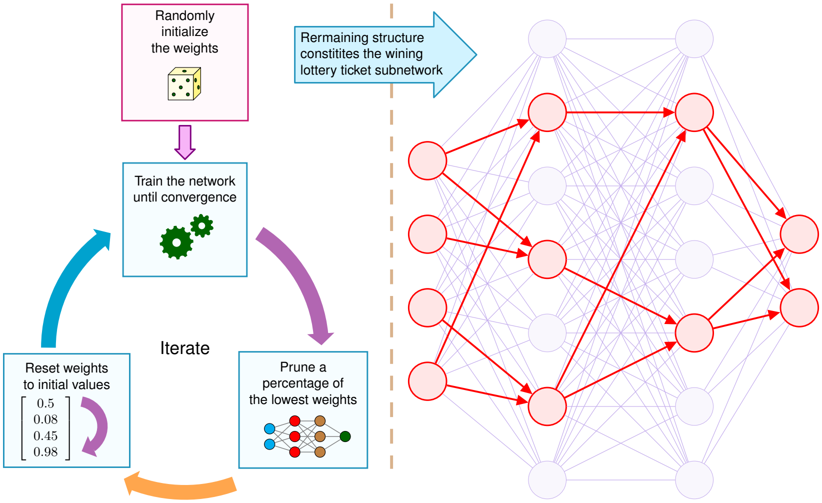

Randomly initialize the weights
Rermaining structure constitites the wining lottery ticket subnetwork
Train the network until convergence
Iterate
Reset weights to initial values
0.5
0.08
0.45
0.98
Prune a percentage of the lowest weights

Figure 10.8: The lottery ticket hypothesis.

The implications of the Lottery Ticket Hypothesis extend beyond conventional pruning techniques. Instead of training large models and pruning them later, LTH suggests that compact, high-performing subnetworks could be trained directly from the start, eliminating the need for overparameterization. This insight challenges the traditional assumption that model size is necessary for effective learning. It also emphasizes the importance of initialization, as winning tickets only retain their performance when reset to their original weight values. This finding raises deeper questions about the role of initialization in shaping a network’s learning trajectory.

The hypothesis further reinforces the effectiveness of iterative pruning over one-shot pruning. Gradually refining the model structure allows the network

[9] Small, well-initialized subnetworks within larger models that, when trained alone, achieve or exceed the full model’s accuracy.

to adapt at each stage, preserving accuracy more effectively than removing large portions of the model in a single step. This process aligns well with practical pruning strategies used in deployment, where preserving accuracy while reducing computation is critical.

Despite its promise, applying LTH in practice remains computationally expensive, as identifying winning tickets requires multiple cycles of pruning and retraining. Ongoing research explores whether winning subnetworks can be detected early without full training, potentially leading to more efficient sparse training techniques. If such methods become practical, LTH could fundamentally reshape how machine learning models are trained, shifting the focus from pruning large networks after training to discovering and training only the essential components from the beginning.

While LTH presents a compelling theoretical perspective on pruning, practical implementations rely on established framework-level tools to integrate structured and unstructured pruning techniques.

10.4.1.9 Pruning Practice

Several machine learning frameworks provide built-in tools to apply structured and unstructured pruning, fine-tune pruned models, and optimize deployment for cloud, edge, and mobile environments.

Machine learning frameworks such as PyTorch, TensorFlow, and ONNX offer dedicated pruning utilities that allow practitioners to efficiently implement these techniques while ensuring compatibility with deployment hardware.

In PyTorch, pruning is available through the `torch.nn.utils.prune` module, which provides functions to apply magnitude-based pruning to individual layers or the entire model. Users can perform unstructured pruning by setting a fraction of the smallest-magnitude weights to zero or apply structured pruning to remove entire neurons or filters. PyTorch also allows for custom pruning strategies, where users define pruning criteria beyond weight magnitude, such as activation-based or gradient-based pruning. Once a model is pruned, it can be fine-tuned to recover lost accuracy before being exported for inference.

TensorFlow provides pruning support through the TensorFlow Model Optimization Toolkit (TF-MOT). This toolkit integrates pruning directly into the training process by applying sparsity-inducing regularization. TensorFlow's pruning API supports global and layer-wise pruning, dynamically selecting parameters for removal based on weight magnitudes. Unlike PyTorch, TensorFlow's pruning is typically applied during training, allowing models to learn sparse representations from the start rather than pruning them post-training. TF-MOT also provides export tools to convert pruned models into TFLite format, making them compatible with mobile and edge devices.

ONNX, an open standard for model representation, does not implement pruning directly but provides export and compatibility support for pruned models from PyTorch and TensorFlow. Since ONNX is designed to be hardware-agnostic, it allows models that have undergone pruning in different frameworks to be optimized for inference engines such as TensorRT, OpenVINO, and EdgeTPU. These inference engines can further leverage structured and dynamic pruning for execution efficiency, particularly on specialized hardware accelerators.

Although framework-level support for pruning has advanced significantly, applying pruning in practice requires careful consideration of hardware compatibility and software optimizations. Standard CPUs and GPUs often do not natively accelerate sparse matrix operations, meaning that unstructured pruning may reduce memory usage without providing significant computational speed-ups. In contrast, structured pruning is more widely supported in inference engines, as it directly reduces the number of computations needed during execution. Dynamic pruning, when properly integrated with inference engines, can optimize execution based on workload variations and hardware constraints, making it particularly beneficial for adaptive AI applications.

At a practical level, choosing the right pruning strategy depends on several key trade-offs, including memory efficiency, computational performance, accuracy retention, and implementation complexity. These trade-offs impact how pruning methods are applied in real-world machine learning workflows, influencing deployment choices based on resource constraints and system requirements.

To help guide these decisions, Table 10.3 provides a high-level comparison of these trade-offs, summarizing the key efficiency and usability factors that practitioners must consider when selecting a pruning method.

Table 10.3: Comparison of pruning strategies.

Criterion	Unstructured Pruning	Structured Pruning	Dynamic Pruning
Memory Efficiency	↑↑ High	↑ Moderate	↑ Moderate
Computational Efficiency	→ Neutral	↑↑ High	↑ High
Accuracy Retention	↑ Moderate	↓↓ Low	↑↑ High
Hardware Compatibility	↓ Low	↑↑ High	→ Neutral
Implementation Complexity	→ Neutral	↑ Moderate	↓↓ High

These trade-offs underscore the importance of aligning pruning methods with practical deployment needs. Frameworks such as PyTorch, TensorFlow, and ONNX enable developers to implement these strategies, but the effectiveness of a pruning approach depends on the underlying hardware and application requirements.

For example, structured pruning is commonly used in mobile and edge applications because of its compatibility with standard inference engines, whereas dynamic pruning is better suited for adaptive AI workloads that need to adjust sparsity levels on the fly. Unstructured pruning, while useful for reducing memory footprints, requires specialized sparse execution kernels to fully realize computational savings.

Understanding these trade-offs is essential when deploying pruned models in real-world settings. Several high-profile models have successfully integrated pruning to optimize performance. MobileNet, a lightweight convolutional neural network designed for mobile and embedded applications, has been pruned to reduce inference latency while preserving accuracy (A. G. Howard et al. 2017b). BERT, a widely used transformer model for natural language processing, has undergone structured pruning of attention heads and intermediate layers to create efficient versions such as DistilBERT and TinyBERT, which retain much of the original performance while reducing computational overhead

(Sanh et al. 2019). In computer vision, EfficientNet has been pruned to remove unnecessary filters, optimizing it for deployment in resource-constrained environments (Tan and Le 2019a).

10.4.2 Knowledge Distillation

Machine learning models are often trained with the goal of achieving the highest possible accuracy, leading to the development of large, complex architectures with millions or even billions of parameters. While these models excel in performance, they are computationally expensive and difficult to deploy in resource-constrained environments such as mobile devices, edge computing platforms, and real-time inference systems. Knowledge distillation is a technique designed to transfer the knowledge of a large, high-capacity model (the teacher) into a smaller, more efficient model (the student) while preserving most of the original model's performance (Gou et al. 2021).

Unlike pruning, which removes unnecessary parameters from a trained model, knowledge distillation involves training a separate, smaller model using guidance from a larger pre-trained model. The student model does not simply learn from labeled data but instead is optimized to match the soft predictions of the teacher model (Jiong Lin et al. 2020). These soft targets, which are probability distributions over classes rather than hard labels, contain richer information about how the teacher model generalizes beyond just the correct answer, helping the student learn more efficiently.

As illustrated in Figure 10.9, the knowledge distillation process involves two models: a high-capacity teacher model (top) and a smaller student model (bottom). The teacher model is first trained on the given dataset and produces a probability distribution over classes using a softened softmax function with temperature T. These soft labels encode more information than traditional hard labels by capturing the relative similarities between different classes. The student model is trained using both these soft labels and the ground truth hard labels.

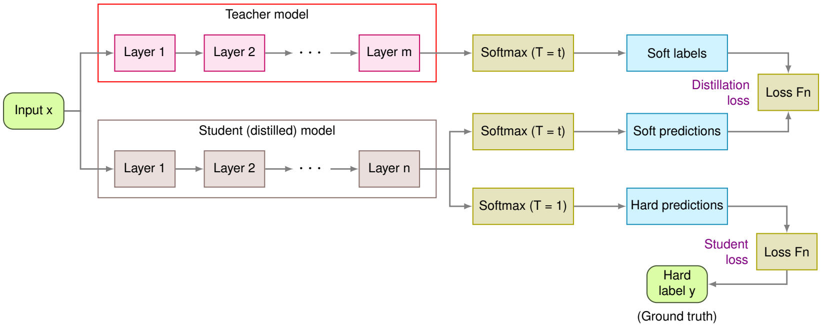


Figure 10.9: Knowledge distillation.

The training process for the student model incorporates two loss terms:

- **Distillation loss**: A loss function (often based on Kullback-Leibler (KL) divergence) that minimizes the difference between the student's and teacher's soft label distributions.

- **Student loss**: A standard cross-entropy loss that ensures the student model correctly classifies the hard labels.

The combination of these two loss functions enables the student model to absorb both structured knowledge from the teacher and label supervision from the dataset. This approach allows smaller models to reach accuracy levels close to their larger teacher models, making knowledge distillation a key technique for model compression and efficient deployment.

Knowledge distillation allows smaller models to reach a level of accuracy that would be difficult to achieve through standard training alone. This makes it particularly useful in ML systems where inference efficiency is a priority, such as real-time applications, cloud-to-edge model compression, and low-power AI systems (Sun et al. 2019).

10.4.2.1 Distillation Theory

Knowledge distillation is based on the idea that a well-trained teacher model encodes more information about the data distribution than just the correct class labels. In conventional supervised learning, a model is trained to minimize the cross-entropy loss[10] between its predictions and the ground truth labels. However, this approach only provides a hard decision boundary for each class, discarding potentially useful information about how the model relates different classes to one another (Hinton, Vinyals, and Dean 2015b).

In contrast, knowledge distillation transfers this additional information by using the soft probability distributions produced by the teacher model. Instead of training the student model to match only the correct label, it is trained to match the teacher's full probability distribution over all possible classes. This is achieved by introducing a temperature-scaled softmax function[11], which smooths the probability distribution, making it easier for the student model to learn from the teacher's outputs (Gou et al. 2021).

10.4.2.2 Distillation Mathematics

Let z_i be the logits (pre-softmax outputs) of the model for class i. The standard softmax function computes class probabilities as:

$$p_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

where higher logits correspond to higher confidence in a class prediction.

In knowledge distillation, we introduce a temperature parameter T that scales the logits before applying softmax:

$$p_i(T) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

where a higher temperature produces a softer probability distribution, revealing more information about how the model distributes uncertainty across different classes.

[10] Cross-entropy loss: A loss function used to measure the difference between two probability distributions.

[11] Softmax: A function that converts logits into probabilities by scaling them based on a temperature parameter.

The student model is then trained using a loss function that minimizes the difference between its output distribution and the teacher's softened output distribution. The most common formulation combines two loss terms:

$$\mathcal{L}_{\text{distill}} = (1-\alpha)\mathcal{L}_{\text{CE}}(y_s, y) + \alpha T^2 \sum_i p_i^T \log p_{i,s}^T$$

where:

- $\mathcal{L}_{\text{CE}}(y_s, y)$ is the standard cross-entropy loss between the student's predictions y_s and the ground truth labels y.
- The second term minimizes the Kullback-Leibler (KL) divergence between the teacher's softened predictions p_i^T and the student's predictions $p_{i,s}^T$.
- The factor T^2 ensures that gradients remain appropriately scaled when using high-temperature values.
- The hyperparameter α balances the importance of the standard training loss versus the distillation loss.

By learning from both hard labels and soft teacher outputs, the student model benefits from the generalization power of the teacher, improving its ability to distinguish between similar classes even with fewer parameters.

10.4.2.3 Distillation Intuition

By learning from both hard labels[12] and soft teacher outputs, the student model benefits from the generalization power of the teacher, improving its ability to distinguish between similar classes even with fewer parameters. Unlike conventional training, where a model learns only from binary correctness signals, knowledge distillation allows the student to absorb a richer understanding of the data distribution from the teacher's predictions.

A key advantage of soft targets is that they provide relative confidence levels rather than just a single correct answer. Consider an image classification task where the goal is to distinguish between different animal species. A standard model trained with hard labels will only receive feedback on whether its prediction is right or wrong. If an image contains a cat, the correct label is "cat," and all other categories, such as "dog" and "fox," are treated as equally incorrect. However, a well-trained teacher model naturally understands that a cat is more visually similar to a dog than to a fox, and its soft output probabilities might look like Figure 10.10, where the relative confidence levels indicate that while "cat" is the most likely category, "dog" is still a plausible alternative, whereas "fox" is much less likely.

Rather than simply forcing the student model to classify the image strictly as a cat, the teacher model provides a more nuanced learning signal, indicating that while "dog" is incorrect, it is a more reasonable mistake than "fox." This subtle information helps the student model build better decision boundaries between similar classes, making it more robust to ambiguity in real-world data.

This effect is particularly useful in cases where training data is limited or noisy. A large teacher model trained on extensive data has already learned to generalize well, capturing patterns that might be difficult to discover with smaller datasets. The student benefits by inheriting this structured knowledge,

[12] Hard Labels: Binary indications of whether a prediction is correct, used in traditional supervised learning.

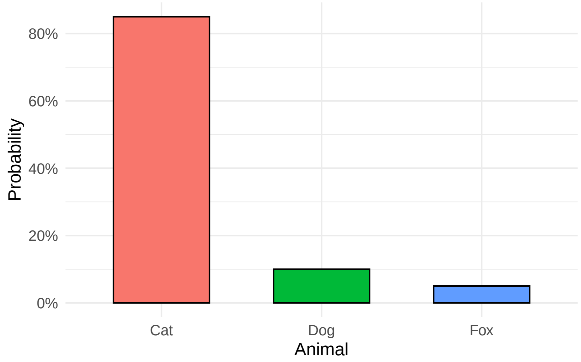


Figure 10.10: Soft target probability distribution.

acting as if it had access to a larger training signal than what is explicitly available.

Another key benefit of knowledge distillation is its regularization effect. Because soft targets distribute probability mass across multiple classes, they prevent the student model from overfitting to specific hard labels. Instead of confidently assigning a probability of 1.0 to the correct class and 0.0 to all others, the student learns to make more calibrated predictions, which improves its generalization performance. This is especially important when the student model has fewer parameters, as smaller networks are more prone to overfitting.

Finally, distillation helps compress large models into smaller, more efficient versions without major performance loss. Training a small model from scratch often results in lower accuracy because the model lacks the capacity to learn the complex representations that a larger network can capture. However, by leveraging the knowledge of a well-trained teacher, the student can reach a higher accuracy than it would have on its own, making it a more practical choice for real-world ML deployments, particularly in edge computing, mobile applications, and other resource-constrained environments.

10.4.2.4 Efficiency Gains

Knowledge distillation is widely used in machine learning systems because it enables smaller models to achieve performance levels comparable to larger models, making it an essential technique for optimizing inference efficiency. While pruning reduces the size of a trained model by removing unnecessary parameters, knowledge distillation improves efficiency by training a compact model from the start, leveraging the teacher's guidance to enhance learning (Sanh et al. 2019). This allows the student model to reach a level of accuracy that would be difficult to achieve through standard training alone.

The efficiency benefits of knowledge distillation can be categorized into three key areas: memory efficiency, computational efficiency, and deployment flexibility.

Memory and Model Compression. A key advantage of knowledge distillation is that it enables smaller models to retain much of the predictive power of larger models, significantly reducing memory footprint. This is particularly useful in resource-constrained environments such as mobile and embedded AI systems, where model size directly impacts storage requirements and load times.

For instance, models such as DistilBERT in NLP and MobileNet distillation variants in computer vision have been shown to retain up to 97% of the accuracy of their larger teacher models while using only half the number of parameters. This level of compression is often superior to pruning, where aggressive parameter reduction can lead to deterioration in representational power.

Another key benefit of knowledge distillation is its ability to transfer robustness and generalization from the teacher to the student. Large models are often trained with extensive datasets and develop strong generalization capabilities, meaning they are less sensitive to noise and data shifts. A well-trained student model inherits these properties, making it less prone to overfitting and more stable across diverse deployment conditions. This is particularly useful in low-data regimes, where training a small model from scratch may result in poor generalization due to insufficient training examples.

Computation and Inference Speed. By training the student model to approximate the teacher's knowledge in a more compact representation, distillation results in models that require fewer FLOPs per inference, leading to faster execution times. Unlike unstructured pruning, which may require specialized hardware support for sparse computation, a distilled model remains densely structured, making it more compatible with existing machine learning accelerators such as GPUs, TPUs, and edge AI chips (Jiao et al. 2020).

In real-world deployments, this translates to:

- Reduced inference latency, which is important for real-time AI applications such as speech recognition, recommendation systems, and self-driving perception models.
- Lower energy consumption, making distillation particularly relevant for low-power AI on mobile devices and IoT systems.
- Higher throughput in cloud inference, where serving a distilled model allows large-scale AI applications to reduce computational cost while maintaining model quality.

For example, when deploying transformer models for NLP, organizations often use teacher-student distillation to create models that achieve similar accuracy at 2-4× lower latency, making it feasible to serve billions of requests per day with significantly lower computational overhead.

Deployment and System Considerations. Knowledge distillation is also effective in multi-task learning[13] scenarios, where a single teacher model can guide multiple student models for different tasks. For example, in multi-lingual NLP models, a large teacher trained on multiple languages can transfer language-specific knowledge to smaller, task-specific student models, enabling efficient deployment across different languages without retraining from scratch. Similarly, in computer vision, a teacher trained on diverse object categories can distill knowledge into specialized students optimized for tasks such as face recognition, medical imaging, or autonomous driving.

Once a student model is distilled, it can be further optimized for hardware-specific acceleration using techniques such as pruning, quantization, and graph

[13] Multi-task Learning: A learning paradigm where a model learns multiple tasks simultaneously, improving generalization.

optimization. This ensures that compressed models remain inference-efficient across multiple hardware environments, particularly in edge AI and mobile deployments (Gordon, Duh, and Andrews 2020).

Despite its advantages, knowledge distillation has some limitations. The effectiveness of distillation depends on the quality of the teacher model—a poorly trained teacher may transfer incorrect biases to the student. Additionally, distillation introduces an additional training phase, where both the teacher and student must be used together, increasing computational costs during training. In some cases, designing an appropriate student model architecture that can fully benefit from the teacher's knowledge remains a challenge, as overly small student models may not have enough capacity to absorb all the relevant information.

10.4.2.5 Trade-offs

Knowledge distillation is a powerful technique for compressing large models into smaller, more efficient versions while maintaining accuracy. By training a student model under the supervision of a teacher model, distillation enables better generalization and inference efficiency compared to training a small model from scratch. It is particularly effective in low-resource environments, such as mobile devices, edge AI, and large-scale cloud inference, where balancing accuracy, speed, and memory footprint is essential.

Compared to pruning, distillation preserves accuracy better but comes at the cost of higher training complexity, as it requires training a new model instead of modifying an existing one. However, pruning provides a more direct computational efficiency gain, especially when structured pruning is used. In practice, combining pruning and distillation often yields the best trade-off, as seen in models like DistilBERT and MobileBERT, where pruning first reduces unnecessary parameters before distillation optimizes a final student model. Table 10.4 summarizes the key trade-offs between knowledge distillation and pruning.

Table 10.4: Comparison of knowledge distillation and pruning.

Criterion	Knowledge Distillation	Pruning
Accuracy retention	High – Student learns from teacher, better generalization	Varies – Can degrade accuracy if over-pruned
Training cost	Higher – Requires training both teacher and student	Lower – Only fine-tuning needed
Inference speed	High – Produces dense, optimized models	Depends – Structured pruning is efficient, unstructured needs special support
Hardware compatibility	High – Works on standard accelerators	Limited – Sparse models may need specialized execution
Ease of implementation	Complex – Requires designing a teacher-student pipeline	Simple – Applied post-training

Knowledge distillation remains an essential technique in ML systems optimization, often used alongside pruning and quantization for deployment-ready models. The next section explores quantization, a method that further reduces computational cost by lowering numerical precision.

10.4.3 Structured Approximations

Machine learning models often contain a significant degree of parameter redundancy, leading to inefficiencies in computation, storage, and energy consumption. The preceding sections on pruning and knowledge distillation introduced methods that explicitly remove redundant parameters or transfer knowledge to a smaller model. In contrast, approximation-based compression techniques focus on restructuring model representations to reduce complexity while maintaining expressive power.

Rather than eliminating individual parameters, approximation methods decompose large weight matrices and tensors into lower-dimensional components, allowing models to be stored and executed more efficiently. These techniques leverage the observation that many high-dimensional representations can be well-approximated by lower-rank structures, thereby reducing the number of parameters without a substantial loss in performance. Unlike pruning, which selectively removes connections, or distillation, which transfers learned knowledge, factorization-based approaches optimize the internal representation of a model through structured approximations.

Among the most widely used approximation techniques are:

- **Low-Rank Matrix Factorization (LRMF)**: A method for decomposing weight matrices into products of lower-rank matrices, reducing storage and computational complexity.
- **Tensor Decomposition**: A generalization of LRMF to higher-dimensional tensors, enabling more efficient representations of multi-way interactions in neural networks.

These methods have been widely applied in machine learning to improve model efficiency, particularly in resource-constrained environments such as edge ML and Tiny ML. Additionally, they play a key role in accelerating model training and inference by reducing the number of required operations. The following sections will provide a detailed examination of low-rank matrix factorization and tensor decomposition, including their mathematical foundations, applications, and associated trade-offs.

10.4.3.1 Low-Rank Factorization

Many machine learning models contain a significant degree of redundancy in their weight matrices, leading to inefficiencies in computation, storage, and deployment. In the previous sections, pruning and knowledge distillation were introduced as methods to reduce model size—pruning by selectively removing parameters and distillation by transferring knowledge from a larger model to a smaller one. However, these techniques do not fundamentally alter the structure of the model's parameters. Instead, they focus on reducing redundant weights or optimizing training processes.

Low-Rank Matrix Factorization (LRMF) provides an alternative approach by approximating a model's weight matrices with lower-rank representations, rather than explicitly removing or transferring information. This technique restructures large parameter matrices into compact, lower-dimensional components, preserving most of the original information while significantly reducing storage and computational costs. Unlike pruning, which creates sparse representations, or distillation, which requires an additional training process, LRMF is a purely mathematical transformation that decomposes a weight matrix into two or more smaller matrices.

This structured compression is particularly useful in machine learning systems where efficiency is a primary concern, such as edge computing, cloud inference, and hardware-accelerated ML execution. By leveraging low-rank approximations, models can achieve substantial reductions in parameter storage while maintaining predictive accuracy, making LRMF a valuable tool for optimizing machine learning architectures.

Training Mathematics. LRMF is a mathematical technique used in linear algebra and machine learning systems to approximate a high-dimensional matrix by decomposing it into the product of lower-dimensional matrices. This factorization enables a more compact representation of model parameters, reducing both memory footprint and computational complexity while preserving essential structural information. In the context of machine learning systems, LRMF plays a crucial role in optimizing model efficiency, particularly for resource-constrained environments such as edge AI and embedded deployments.

Formally, given a matrix $A \in \mathbb{R}^{m \times n}$, LRMF seeks two matrices $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{k \times n}$ such that:

$$A \approx UV$$

where k is the rank of the approximation, typically much smaller than both m and n. This approximation is commonly obtained through singular value decomposition (SVD), where A is factorized as:

$$A = U \Sigma V^T$$

where Σ is a diagonal matrix containing singular values, and U and V are orthogonal matrices. By retaining only the top k singular values, a low-rank approximation of A is obtained.

Figure 10.11 illustrates the decrease in parameterization enabled by low-rank matrix factorization. Observe how the matrix M can be approximated by the product of matrices L_k and R_k^T. For intuition, most fully connected layers in networks are stored as a projection matrix M, which requires $m \times n$ parameters to be loaded during computation. However, by decomposing and approximating it as the product of two lower-rank matrices, we only need to store $m \times k + k \times n$ parameters in terms of storage while incurring an additional compute cost of the matrix multiplication. So long as $k < n/2$, this factorization has fewer total parameters to store while adding a computation of runtime $O(mkn)$ (Gu 2023).

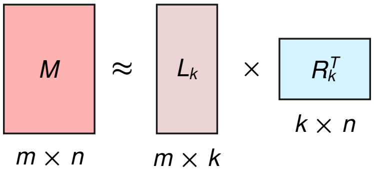


Figure 10.11: Low matrix factorization. Source: The Clever Machine.

LRMF is widely used to enhance the efficiency of machine learning models by reducing parameter redundancy, particularly in fully connected and convolutional layers. In the broader context of machine learning systems, factorization techniques contribute to optimizing model inference speed, storage efficiency, and adaptability to specialized hardware accelerators.

Fully connected layers often contain large weight matrices, making them ideal candidates for factorization. Instead of storing a dense $m \times n$ weight matrix, LRMF allows for a more compact representation with two smaller matrices of dimensions $m \times k$ and $k \times n$, significantly reducing storage and computational costs. This reduction is particularly valuable in cloud-to-edge ML pipelines, where minimizing model size can facilitate real-time execution on embedded devices.

Convolutional layers can also benefit from LRMF by decomposing convolutional filters into separable structures. Techniques such as depthwise-separable convolutions leverage factorization principles to achieve computational efficiency without significant loss in accuracy. These methods align well with hardware-aware optimizations used in modern AI acceleration frameworks.

LRMF has been extensively used in collaborative filtering for recommendation systems. By factorizing user-item interaction matrices, latent factors corresponding to user preferences and item attributes can be extracted, enabling efficient and accurate recommendations. Within large-scale machine learning systems, such optimizations directly impact scalability and performance in production environments.

Factorization Efficiency and Challenges. By factorizing a weight matrix into lower-rank components, the number of parameters required for storage is reduced from $O(mn)$ to $O(mk+kn)$, where k is significantly smaller than m, n. However, this reduction comes at the cost of an additional matrix multiplication operation during inference, potentially increasing computational latency. In machine learning systems, this trade-off is carefully managed to balance storage efficiency and real-time inference speed.

Choosing an appropriate rank k is a key challenge in LRMF. A smaller k results in greater compression but may lead to significant information loss, while a larger k retains more information but offers limited efficiency gains. Methods such as cross-validation and heuristic approaches are often employed to determine the optimal rank, particularly in large-scale ML deployments where compute and storage constraints vary.

In real-world machine learning applications, datasets may contain noise or missing values, which can affect the quality of factorization. Regularization techniques, such as adding an L_2 penalty, can help mitigate overfitting and improve the robustness of LRMF, ensuring stable performance across different ML system architectures.

Low-rank matrix factorization provides an effective approach for reducing the complexity of machine learning models while maintaining their expressive power. By approximating weight matrices with lower-rank representations, LRMF facilitates efficient inference and model deployment, particularly in resource-constrained environments such as edge computing. Within machine learning systems, factorization techniques contribute to scalable, hardware-aware optimizations that enhance real-world model performance. Despite challenges such as rank selection and computational overhead, LRMF remains a valuable tool for improving efficiency in ML system design and deployment.

10.4.3.2 Tensor Decomposition

While low-rank matrix factorization provides an effective method for compressing large weight matrices in machine learning models, many modern architectures rely on multi-dimensional tensors rather than two-dimensional matrices. Convolutional layers, attention mechanisms, and embedding representations commonly involve multi-way interactions that cannot be efficiently captured using standard matrix factorization techniques. In such cases, tensor decomposition provides a more general approach to reducing model complexity while preserving structural relationships within the data.

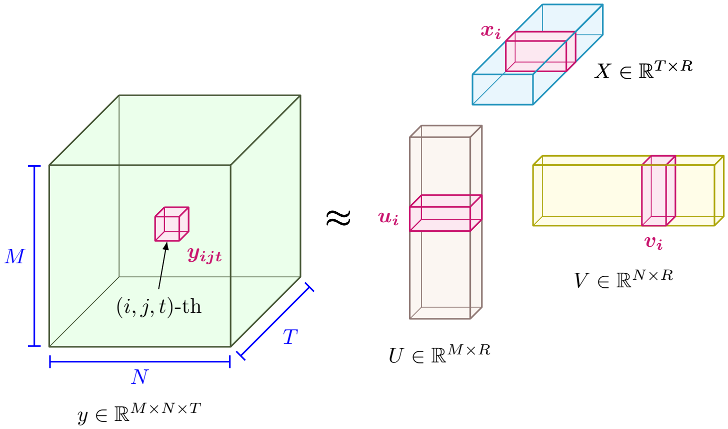

x_i
$X \in \mathbb{R}^{T\times R}$
M
y_{ijt}
(i,j,t)-th
N
T
$y \in \mathbb{R}^{M\times N\times T}$
$\approx$
u_i
$U \in \mathbb{R}^{M\times R}$
v_i
$V \in \mathbb{R}^{N\times R}$

Figure 10.12: Tensor decomposition. Source: Richter and Zhao (2021).

Tensor decomposition (TD) extends the principles of low-rank factorization to higher-order tensors, allowing large multi-dimensional arrays to be expressed in terms of lower-rank components (see Figure 10.12). Given that tensors frequently appear in machine learning systems as representations of weight parameters, activations, and input features, their direct storage and computation often become impractical. By decomposing these tensors into a set of smaller factors, tensor decomposition significantly reduces memory requirements and computational overhead while maintaining the integrity of the original structure.

This approach is widely used in machine learning to improve efficiency across various architectures. In convolutional neural networks, tensor decomposition enables the approximation of convolutional kernels with lower-dimensional factors, reducing the number of parameters while preserving the representational power of the model. In natural language processing, high-dimensional embeddings can be factorized into more compact representations, leading to faster inference and reduced memory consumption. In hardware acceleration, tensor decomposition helps optimize tensor operations for execution on specialized processors, ensuring efficient utilization of computational resources.

Training Mathematics. A tensor is a multi-dimensional extension of a matrix, representing data across multiple axes rather than being confined to two-dimensional structures. In machine learning, tensors naturally arise in various contexts, including the representation of weight parameters, activations, and input features. Given the high dimensionality of these tensors, direct storage and computation often become impractical, necessitating efficient factorization techniques.

Tensor decomposition generalizes the principles of low-rank matrix factorization by approximating a high-order tensor with a set of lower-rank components. Formally, for a given tensor $\mathcal{A} \in \mathbb{R}^{m \times n \times p}$, the goal of decomposition is to express $\mathcal{A}$ in terms of factorized components that require fewer parameters to store and manipulate. This decomposition reduces the memory footprint and computational requirements while retaining the structural relationships present in the original tensor.

Several factorization methods have been developed for tensor decomposition, each suited to different applications in machine learning. One common approach is CANDECOMP/PARAFAC (CP) decomposition, which expresses a tensor as a sum of rank-one components. In CP decomposition, a tensor $\mathcal{A} \in \mathbb{R}^{m \times n \times p}$ is approximated as

$$\mathcal{A} \approx \sum_{r=1}^{k} u_r \otimes v_r \otimes w_r$$

where $u_r \in \mathbb{R}^m$, $v_r \in \mathbb{R}^n$, and $w_r \in \mathbb{R}^p$ are factor vectors and k is the rank of the approximation.

Another widely used approach is Tucker decomposition, which generalizes singular value decomposition to tensors by introducing a core tensor $\mathcal{G} \in \mathbb{R}^{k_1 \times k_2 \times k_3}$ and factor matrices $U \in \mathbb{R}^{m \times k_1}$, $V \in \mathbb{R}^{n \times k_2}$, and $W \in \mathbb{R}^{p \times k_3}$, such that

$$\mathcal{A} \approx \mathcal{G} \times_1 U \times_2 V \times_3 W$$

where $\times_i$ denotes the mode-i tensor-matrix multiplication.

Another method, Tensor-Train (TT) decomposition, factorizes high-order tensors into a sequence of lower-rank matrices, reducing both storage and computational complexity. Given a tensor $\mathcal{A} \in \mathbb{R}^{m_1 \times m_2 \times \cdots \times m_d}$, TT decomposition represents it as a product of lower-dimensional tensor cores $\mathcal{G}^{(i)}$, where each core $\mathcal{G}^{(i)}$ has dimensions $\mathbb{R}^{r_{i-1} \times m_i \times r_i}$, and the full tensor is reconstructed as

$$\mathcal{A} \approx \mathcal{G}^{(1)} \times \mathcal{G}^{(2)} \times \cdots \times \mathcal{G}^{(d)}$$

where r_i are the TT ranks.

These tensor decomposition methods play a crucial role in optimizing machine learning models by reducing parameter redundancy while maintaining expressive power. The next section will examine how these techniques are applied to machine learning architectures and discuss their computational trade-offs.

Tensor Decomposition Applications. Tensor decomposition methods are widely applied in machine learning systems to improve efficiency and scalability. By factorizing high-dimensional tensors into lower-rank representations, these methods reduce memory usage and computational requirements while preserving the model's expressive capacity. This section examines several key applications of tensor decomposition in machine learning, focusing on its impact on convolutional neural networks, natural language processing, and hardware acceleration.

In convolutional neural networks (CNNs), tensor decomposition is used to compress convolutional filters and reduce the number of required operations during inference. A standard convolutional layer contains a set of weight tensors that define how input features are transformed. These weight tensors often exhibit redundancy, meaning they can be decomposed into smaller components without significantly degrading performance. Techniques such as CP decomposition and Tucker decomposition enable convolutional filters to be approximated using lower-rank tensors, reducing the number of parameters and computational complexity of the convolution operation. This form of structured compression is particularly valuable in edge and mobile machine learning applications, where memory and compute resources are constrained.

In natural language processing (NLP), tensor decomposition is commonly applied to embedding layers and attention mechanisms. Many NLP models, including transformers, rely on high-dimensional embeddings to represent words, sentences, or entire documents. These embeddings can be factorized using tensor decomposition to reduce storage requirements without compromising their ability to capture semantic relationships. Similarly, in transformer-based architectures, the self-attention mechanism requires large tensor multiplications, which can be optimized using decomposition techniques to lower the computational burden and accelerate inference.

Hardware acceleration for machine learning also benefits from tensor decomposition by enabling more efficient execution on specialized processors such as GPUs, tensor processing units (TPUs), and field-programmable gate arrays (FPGAs). Many machine learning frameworks include optimizations that leverage tensor decomposition to improve model execution speed and reduce energy consumption. Decomposing tensors into structured low-rank components aligns well with the memory hierarchy of modern hardware accelerators, facilitating more efficient data movement and parallel computation.

Despite these advantages, tensor decomposition introduces certain trade-offs that must be carefully managed. The choice of decomposition method and rank significantly influences model accuracy and computational efficiency. Selecting an overly aggressive rank reduction may lead to excessive information loss, while retaining too many components diminishes the efficiency gains. Additionally, the factorization process itself can introduce a computational overhead, requiring careful consideration when applying tensor decomposition to large-scale machine learning systems.

TD Trade-offs and Challenges. While tensor decomposition provides significant efficiency gains in machine learning systems, it introduces trade-offs that must be carefully managed to maintain model accuracy and computational feasibility. These trade-offs primarily involve the selection of decomposition rank, the computational complexity of factorization, and the stability of factorized representations.

One of the primary challenges in tensor decomposition is determining an appropriate rank for the factorized representation. In low-rank matrix factorization, the rank defines the dimensionality of the factorized matrices, directly influencing the balance between compression and information retention. In tensor decomposition, rank selection becomes even more complex, as different decomposition methods define rank in varying ways. For instance, in CANDECOMP/PARAFAC (CP) decomposition, the rank corresponds to the number of rank-one tensors used to approximate the original tensor. In Tucker decomposition, the rank is determined by the dimensions of the core tensor, while in Tensor-Train (TT) decomposition, the ranks of the factorized components dictate the level of compression. Selecting an insufficient rank can lead to excessive information loss, degrading the model's predictive performance, whereas an overly conservative rank reduction results in limited compression benefits.

Another key challenge is the computational overhead associated with performing tensor decomposition. The factorization process itself requires solving an optimization problem, often involving iterative procedures such as alternating least squares (ALS) or stochastic gradient descent (SGD). These methods can be computationally expensive, particularly for large-scale tensors used in machine learning models. Additionally, during inference, the need to reconstruct tensors from their factorized components introduces additional matrix and tensor multiplications, which may increase computational latency. The efficiency of tensor decomposition in practice depends on striking a balance between reducing parameter storage and minimizing the additional computational cost incurred by factorized representations.

Numerical stability is another concern when applying tensor decomposition to machine learning models. Factorized representations can suffer from numerical instability, particularly when the original tensor contains highly correlated structures or when decomposition methods introduce ill-conditioned factors. Regularization techniques, such as adding constraints on factor matrices or applying low-rank approximations incrementally, can help mitigate these issues. Additionally, the optimization process used for decomposition must be carefully tuned to avoid convergence to suboptimal solutions that fail to preserve the essential properties of the original tensor.

Despite these challenges, tensor decomposition remains a valuable tool for optimizing machine learning models, particularly in applications where reducing memory footprint and computational complexity is a priority. Advances in adaptive decomposition methods, automated rank selection strategies, and hardware-aware factorization techniques continue to improve the practical utility of tensor decomposition in machine learning. The following section will summarize the key insights gained from low-rank matrix factorization and tensor decomposition, highlighting their role in designing efficient machine learning systems.

LRMF vs. TD. Both low-rank matrix factorization and tensor decomposition serve as fundamental techniques for reducing the complexity of machine learning models by approximating large parameter structures with lower-rank representations. While they share the common goal of improving storage efficiency and computational performance, their applications, computational trade-offs, and structural assumptions differ significantly. This section provides a comparative analysis of these two techniques, highlighting their advantages, limitations, and practical use cases in machine learning systems.

One of the key distinctions between LRMF and tensor decomposition lies in the dimensionality of the data they operate on. LRMF applies to two-dimensional matrices, making it particularly useful for compressing weight matrices in fully connected layers or embeddings. Tensor decomposition, on the other hand, extends factorization to multi-dimensional tensors, which arise naturally in convolutional layers, attention mechanisms, and multi-modal learning. This generalization allows tensor decomposition to exploit additional structural properties of high-dimensional data that LRMF cannot capture.

Computationally, both methods introduce trade-offs between storage savings and inference speed. LRMF reduces the number of parameters in a model by factorizing a weight matrix into two smaller matrices, thereby reducing memory footprint while incurring an additional matrix multiplication during inference. In contrast, tensor decomposition further reduces storage by decomposing tensors into multiple lower-rank components, but at the cost of more complex tensor contractions, which may introduce higher computational overhead. The choice between these methods depends on whether the primary constraint is memory storage or inference latency.

Table 10.5 summarizes the key differences between LRMF and tensor decomposition:

Table 10.5: Comparing LRMF with tensor decomposition.

Feature	Low-Rank Matrix Factorization (LRMF)	Tensor Decomposition
Applicable Data Structure	Two-dimensional matrices	Multi-dimensional tensors
Compression Mechanism	Factorizes a matrix into two or more lower-rank matrices	Decomposes a tensor into multiple lower-rank components
Common Methods	Singular Value Decomposition (SVD), Alternating Least Squares (ALS)	CP Decomposition, Tucker Decomposition, Tensor-Train (TT)
Computational Complexity	Generally lower, often $ O(mnk) $ for a rank-$ k $ approximation	Higher, due to iterative optimization and tensor contractions
Storage Reduction	Reduces storage from $ O(mn) $ to $ O(mk + kn) $	Achieves higher compression but requires more complex storage representations
Inference Overhead	Requires additional matrix multiplication	Introduces additional tensor operations, potentially increasing inference latency
Primary Use Cases	Fully connected layers, embeddings, recommendation systems	Convolutional filters, attention mechanisms, multi-modal learning
Implementation Complexity	Easier to implement, often involves direct factorization methods	More complex, requiring iterative optimization and rank selection

Despite these differences, LRMF and tensor decomposition are not mutually exclusive. In many machine learning models, both methods can be applied together to optimize different components of the architecture. For example, fully connected layers may be compressed using LRMF, while convolutional kernels and attention tensors undergo tensor decomposition. The choice of technique ultimately depends on the specific characteristics of the model and the trade-offs between storage efficiency and computational complexity.

10.4.4 Neural Architecture Search

The techniques discussed in previous sections, such as pruning, knowledge distillation, and various other methods, rely on human expertise to determine optimal model configurations. While these manual approaches have led to significant advancements, they are inherently slow, resource-intensive, and constrained by human biases. Selecting an optimal architecture requires extensive experimentation, and even experienced practitioners may overlook more efficient designs (Elsken, Metzen, and Hutter 2019a).

NAS addresses these limitations by automating model design. As illustrated in Figure 10.13, instead of manually tuning configurations, NAS systematically explores a large space of architectures to identify those that best balance accuracy, computational cost, memory efficiency, and inference latency. By framing model selection as a structured search problem, NAS reduces reliance on trial and error, allowing architectures to be discovered programmatically rather than heuristically (Zoph and Le 2017a).

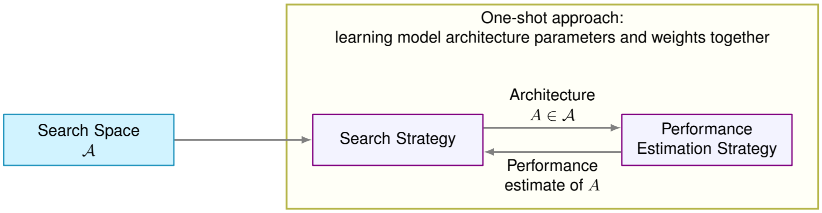

One-shot approach:
learning model architecture parameters and weights together
Search Space
$\mathcal{A}$
Search Strategy
Architecture
$A \in \mathcal{A}$
Performance Estimation Strategy
Performance estimate of A

Figure 10.13: An example flow of neural architecture search, where model architectures and weights are learned together.

NAS formalizes model design as an optimization problem, leveraging techniques such as reinforcement learning, evolutionary algorithms, and gradient-based methods to automate decisions traditionally made by experts (Real et al. 2019a). This approach integrates principles of scaling optimization, structural pruning, and compressed representations, offering a unified framework for model efficiency.

Real-world applications demonstrate that NAS-generated architectures often match or surpass human-designed models in efficiency and accuracy. Examples include models optimized for mobile and cloud environments, where inference latency and memory constraints are critical considerations. Ultimately, NAS encapsulates a holistic approach to model optimization, unifying multiple strategies into an automated, scalable framework.

10.4.4.1 Model Efficiency Encoding

NAS operates in three key stages: defining the search space, exploring candidate architectures, and evaluating their performance. The search space defines the architectural components and constraints that NAS can modify. The search strategy determines how NAS explores possible architectures, selecting promising candidates based on past observations. The evaluation process ensures that the discovered architectures satisfy multiple objectives, including accuracy, efficiency, and hardware suitability.

1. Search Space Definition: This stage establishes the architectural components and constraints NAS can modify, such as the number of layers, con-

volution types, activation functions, and hardware-specific optimizations. A well-defined search space balances innovation with computational feasibility.

2. Search Strategy: NAS explores the search space using methods such as reinforcement learning[14], evolutionary algorithms[15], or gradient-based techniques. These approaches guide the search toward architectures that maximize performance while meeting resource constraints.

3. Evaluation Criteria: Candidate architectures are assessed based on multiple metrics, including accuracy, FLOPs, memory consumption, inference latency, and power efficiency. NAS ensures that the selected architectures align with deployment requirements.

NAS unifies structural design and optimization into a singular, automated framework. The result is the discovery of architectures that are not only highly accurate but also computationally efficient and well-suited for target hardware platforms.

10.4.4.2 Search Space Definition

The first step in NAS is determining the set of architectures it is allowed to explore, known as the search space. The size and structure of this space directly affect how efficiently NAS can discover optimal models. A well-defined search space must be broad enough to allow innovation while remaining narrow enough to prevent unnecessary computation on impractical designs.

A typical NAS search space consists of modular building blocks that define the structure of the model. These include the types of layers available for selection, such as standard convolutions, depthwise separable convolutions,[16] attention mechanisms, and residual blocks.[17] The search space also defines constraints on network depth and width, specifying how many layers the model can have and how many channels each layer should include. Additionally, NAS considers activation functions, such as ReLU, Swish, or GELU, which influence both model expressiveness and computational efficiency.

Other architectural decisions within the search space include kernel sizes, receptive fields, and skip connections, which impact both feature extraction and model complexity. Some NAS implementations also incorporate hardware-aware optimizations, ensuring that the discovered architectures align with specific hardware, such as GPUs, TPUs, or mobile CPUs.

The choice of search space determines the extent to which NAS can optimize a model. If the space is too constrained, the search algorithm may fail to discover novel and efficient architectures. If it is too large, the search becomes computationally expensive, requiring extensive resources to explore a vast number of possibilities. Striking the right balance ensures that NAS can efficiently identify architectures that improve upon human-designed models.

10.4.4.3 Search Space Exploration

Once the search space is defined, NAS must determine how to explore different architectures effectively. The search strategy guides this process by selecting which architectures to evaluate based on past observations. An effective search

[14] Reinforcement Learning: A machine learning technique that learns actions based on rewards.

[15] Evolutionary Algorithms: Optimization algorithms inspired by natural selection.

[16] Depthwise Separable Convolution: A type of convolution that splits the convolutional process into separate spatial and depth-wise steps.

[17] Residual Block: A series of layers in a deep neural network designed to prevent the vanishing gradient problem by adding inputs to outputs.

strategy must balance exploration (testing new architectures) with exploitation (refining promising designs).

Several methods have been developed to navigate the search space efficiently. Reinforcement learning-based NAS formulates the search process as a decision-making problem, where an agent sequentially selects architectural components and receives a reward signal based on the performance of the generated model. Over time, the agent learns to generate better architectures by maximizing this reward. While effective, reinforcement learning-based NAS can be computationally expensive because it requires training many candidate models before converging on an optimal design.

An alternative approach uses evolutionary algorithms, which maintain a population of candidate architectures and iteratively improve them through mutation and selection. Stronger architectures, which possess higher accuracy and efficiency, are retained, while modifications such as changing layer types or filter sizes introduce new variations. This approach has been shown to balance exploration and computational feasibility more effectively than reinforcement learning-based NAS.

More recent methods, such as gradient-based NAS, introduce differentiable parameters that represent architectural choices. Instead of treating architectures as discrete entities, gradient-based methods optimize both model weights and architectural parameters simultaneously using standard gradient descent. This significantly reduces the computational cost of the search, making NAS more practical for real-world applications.

The choice of search strategy has a direct impact on the feasibility of NAS. Early NAS methods that relied on reinforcement learning required weeks of GPU computation to discover a single architecture. More recent methods, particularly those based on gradient-based search, have significantly reduced this cost, making NAS more efficient and accessible.

10.4.4.4 Candidate Architecture Evaluation

Every architecture explored by NAS must be evaluated based on a set of predefined criteria. While accuracy is a fundamental metric, NAS also optimizes for efficiency constraints to ensure that models are practical for deployment. The evaluation process determines whether an architecture should be retained for further refinement or discarded in favor of more promising designs.

The primary evaluation metrics include computational complexity, memory consumption, inference latency, and energy efficiency. Computational complexity, often measured in FLOPs, determines the overall resource demands of a model. NAS favors architectures that achieve high accuracy while reducing unnecessary computations. Memory consumption, which includes both parameter count and activation storage, ensures that models fit within hardware constraints. For real-time applications, inference latency is a key factor, with NAS selecting architectures that minimize execution time on specific hardware platforms. Finally, some NAS implementations explicitly optimize for power consumption, ensuring that models are suitable for mobile and edge devices.

For example, FBNet, a NAS-generated architecture optimized for mobile inference, incorporated latency constraints into the search process. Instead of selecting the most accurate model, NAS identified architectures that provided the best balance between accuracy and inference speed (B. Wu et al. 2019). Similarly, EfficientNet was discovered through NAS by jointly optimizing for accuracy and computational efficiency, resulting in a model that delivers state-of-the-art performance while reducing FLOPs compared to conventional architectures (Tan and Le 2019a).

By integrating these constraints into the search process, NAS systematically discovers architectures that balance accuracy, efficiency, and hardware adaptability. Instead of manually fine-tuning these trade-offs, NAS automates the selection of optimal architectures, ensuring that models are well-suited for real-world deployment scenarios.

10.4.4.5 NAS-Discovered Architecture Examples

NAS has been successfully used to design several state-of-the-art architectures that outperform manually designed models in terms of efficiency and accuracy. These architectures illustrate how NAS integrates scaling optimization, computation reduction, memory efficiency, and hardware-aware design into an automated process.

One of the most well-known NAS-generated models is EfficientNet, which was discovered using a NAS framework that searched for the most effective combination of depth, width, and resolution scaling. Unlike traditional scaling strategies that independently adjust these factors, NAS optimized the model using compound scaling, which applies a fixed set of scaling coefficients to ensure that the network grows in a balanced way. EfficientNet achieves higher accuracy with fewer parameters and lower FLOPs than previous architectures, making it ideal for both cloud and mobile deployment.

Another key example is MobileNetV3, which used NAS to optimize its network structure for mobile hardware. The search process led to the discovery of inverted residual blocks with squeeze-and-excitation layers, which improve accuracy while reducing computational cost. NAS also selected optimized activation functions and efficient depthwise separable convolutions, leading to a $5\times$ reduction in FLOPs compared to earlier MobileNet versions.

FBNet, another NAS-generated model, was specifically optimized for real-time inference on mobile CPUs. Unlike architectures designed for general-purpose acceleration, FBNet's search process explicitly considered latency constraints during training, ensuring that the final model runs efficiently on low-power hardware. Similar approaches have been used in TPU-optimized NAS models, where the search process is guided by hardware-aware cost functions to maximize parallel execution efficiency.

NAS has also been applied beyond convolutional networks. NAS-BERT explores transformer-based architectures, searching for efficient model structures that retain strong natural language understanding capabilities while reducing compute and memory overhead. NAS has been particularly useful in designing efficient vision transformers (ViTs) by automatically discovering lightweight attention mechanisms tailored for edge AI applications.

Each of these NAS-generated models demonstrates how automated architecture search can uncover novel efficiency trade-offs that may not be immediately intuitive to human designers. Explicit encoding of efficiency constraints into the search process enables NAS to systematically produce architectures that are more computationally efficient, memory-friendly, and hardware-adapted than those designed manually (Radosavovic et al. 2020).

10.5 Numerical Precision Optimization

Machine learning models perform computations using numerical representations, and the choice of precision directly affects memory usage, computational efficiency, and power consumption. Many state-of-the-art models are trained and deployed using high-precision floating-point formats, such as FP32 (32-bit floating point), which offer numerical stability and high accuracy (S. Gupta et al. 2015). However, high-precision formats increase storage requirements, memory bandwidth usage, and power consumption, making them inefficient for large-scale or resource-constrained deployments.

Reducing numerical precision improves efficiency by reducing storage needs, decreasing data movement between memory and compute units, and enabling faster computation. Many modern AI accelerators, such as TPUs, GPUs, and edge AI chips, include dedicated hardware for low-precision computation, allowing FP16 and INT8 operations to run at significantly higher throughput than FP32 (Y. E. Wang, Wei, and Brooks 2019). However, reducing precision introduces quantization error, which can lead to accuracy degradation. The extent to which precision can be reduced depends on the model architecture, dataset properties, and hardware support.

This section explores the role of numerical precision in model efficiency, examining the trade-offs between different precision formats, methods for precision reduction, the benefits of custom and adaptive numerical representations, and extreme cases where models operate using only a few discrete numerical states (binarization and ternarization[18]).

10.5.1 Efficiency Numerical Precision

Efficient numerical representations enable significant reductions in storage requirements, computation latency, and power usage. By lowering precision, models can perform inference more efficiently, making this approach particularly beneficial for mobile AI, embedded systems, and cloud inference, where efficiency constraints are paramount. Moreover, efficient numerics facilitate hardware-software co-design, allowing precision levels to be tuned to specific hardware capabilities, thereby maximizing throughput on AI accelerators such as GPUs, TPUs, NPUs, and edge AI chips.[19]

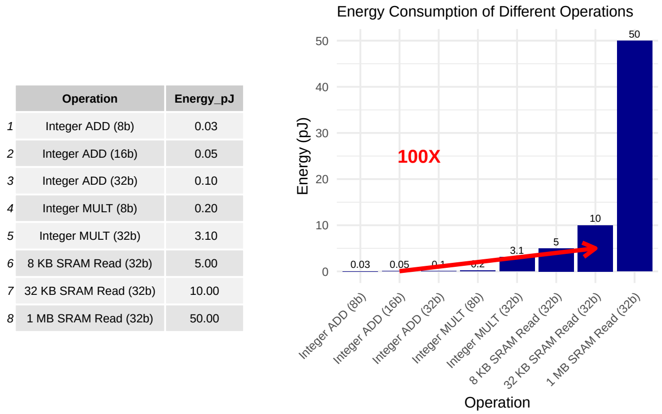
10.5.1.1 Numerical Precision Energy Costs

The energy costs associated with different numerical precisions further highlight the benefits of reducing precision. As shown in Figure 10.14, performing a 32-bit floating-point addition (FAdd)[20] consumes approximately 0.9 pJ, whereas a 16-bit floating-point addition only requires 0.4 pJ. Similarly, a 32-bit integer addition costs 0.1 pJ, while an 8-bit integer addition is significantly lower at

[18] Binarization and Ternarization: Techniques that use 1-bit or 3-state representations to minimize model size and complexity.

[19] Biological neural systems achieve remarkable performance using imprecise, noisy neural signals. This natural efficiency has inspired the development of reduced-precision computational models in artificial neural networks.

[20] Floating-Point Addition (FAdd): An operation to sum two floating-point numbers.



	Operation	Energy_pJ
1	Integer ADD (8b)	0.03
2	Integer ADD (16b)	0.05
3	Integer ADD (32b)	0.10
4	Integer MULT (8b)	0.20
5	Integer MULT (32b)	3.10
6	8 KB SRAM Read (32b)	5.00
7	32 KB SRAM Read (32b)	10.00
8	1 MB SRAM Read (32b)	50.00

Figure 10.14: Coming soon.

just 0.03 pJ. These savings compound when considering large-scale models operating across billions of operations.

Beyond direct compute savings, reducing numerical precision has a significant impact on memory energy consumption, which often dominates total system power. Lower-precision representations reduce data storage requirements and memory bandwidth usage, leading to fewer and more efficient memory accesses. This is critical because accessing memory, particularly off-chip DRAM, is far more energy-intensive than performing arithmetic operations. For instance, DRAM accesses require orders of magnitude more energy (1.3–2.6 nJ) compared to cache accesses (e.g., 10 pJ for an 8 KB L1 cache access). The breakdown of instruction energy further underscores the cost of moving data within the memory hierarchy, where an instruction's total energy can be significantly impacted by memory access patterns.

By reducing numerical precision, models can not only execute computations more efficiently but also reduce data movement, leading to lower overall energy consumption. This is particularly important for hardware accelerators and edge devices, where memory bandwidth and power efficiency are key constraints.

10.5.1.2 Quantization Performance Gains

Figure 10.15 illustrates the impact of quantization on both inference time and model size using a stacked bar chart with a dual-axis representation. The left bars in each category show inference time improvements when moving from FP32 to INT8, while the right bars depict the corresponding reduction in model size. The results indicate that quantized models achieve up to $4\times$ faster inference while reducing storage requirements by a factor of $4\times$, making them highly suitable for deployment in resource-constrained environments.

However, reducing numerical precision introduces trade-offs. Lower-precision formats can lead to numerical instability[21] and quantization noise[22], potentially affecting model accuracy. Some architectures, such as large transformer-based NLP models, tolerate precision reduction well, whereas others may experience significant degradation. Thus, selecting the appropriate numerical precision requires balancing accuracy constraints, hardware support, and efficiency gains.

[21] Numerical Instability: Occurs when small errors in data or operations amplify through a computation, possibly leading to erroneous results.

[22] Quantization Noise: Unwanted variability in model output due to reduced precision in numerical calculations.

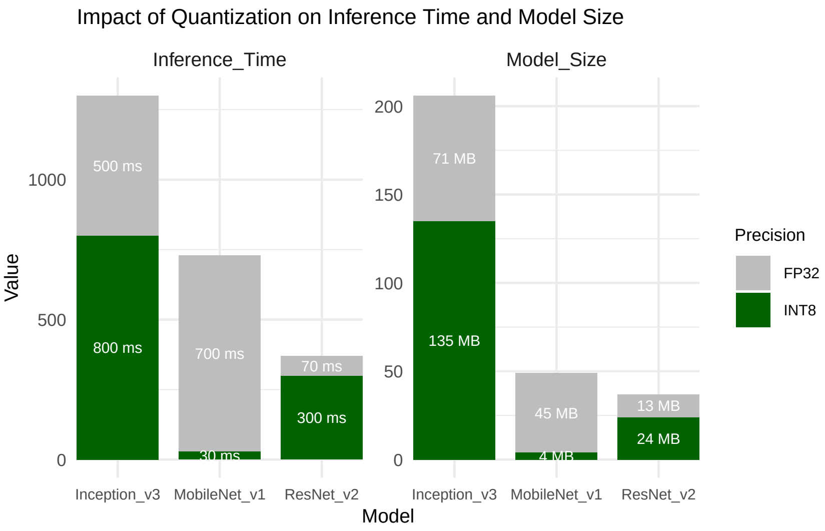


Figure 10.15: Impact of quantization on inference time and model size. The left stacked bars show inference time improvements, while the right stacked bars highlight memory savings.

10.5.1.3 Numerical Precision Reduction Trade-offs

However, reducing numerical precision introduces trade-offs. Lower-precision formats can lead to numerical instability and quantization noise, potentially affecting model accuracy. Some architectures, such as large transformer-based NLP models, tolerate precision reduction well, whereas others may experience significant degradation. Thus, selecting the appropriate numerical precision requires balancing accuracy constraints, hardware support, and efficiency gains.

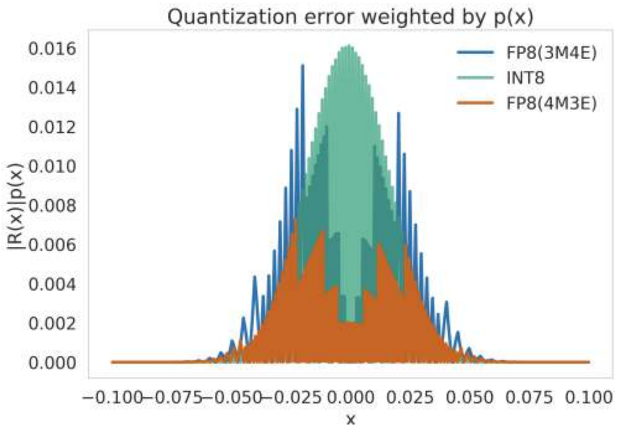


Figure 10.16: Quantization error weighted by p(x).

The figure above illustrates the quantization error weighted by the probability distribution of values, comparing different numerical formats (FP8 variants and INT8). The error distribution highlights how different formats introduce varying levels of quantization noise across the range of values, which in turn influences model accuracy and stability.

10.5.2 Numeric Encoding and Storage

The representation of numerical data in machine learning systems extends beyond precision levels to encompass encoding formats and storage mechanisms, both of which significantly influence computational efficiency. The encoding of numerical values determines how floating-point and integer representations are stored in memory and processed by hardware, directly affecting performance in machine learning workloads. As machine learning models grow in size and complexity, optimizing numeric encoding becomes increasingly critical for ensuring efficiency, particularly on specialized hardware accelerators (Mellempudi et al. 2019).

Floating-point representations, which are widely used in machine learning, follow the IEEE 754 standard, defining how numbers are represented using a combination of sign, exponent, and mantissa (fraction) bits. Standard formats such as FP32 (single precision) and FP64 (double precision) provide high accuracy but demand substantial memory and computational resources. To enhance efficiency, reduced-precision formats such as FP16, bfloat16, and FP8 have been introduced, offering lower storage requirements while maintaining sufficient numerical range for machine learning computations. Unlike FP16, which allocates more bits to the mantissa, bfloat16 retains the same exponent size as FP32, allowing it to represent a wider dynamic range while reducing precision in the fraction. This characteristic makes bfloat16 particularly effective for machine learning training, where maintaining dynamic range is critical for stable gradient updates.

Integer-based representations, including INT8 and INT4, further reduce storage and computational overhead by eliminating the need for exponent and mantissa encoding. These formats are commonly used in quantized inference, where model weights and activations are converted to discrete integer values to accelerate computation and reduce power consumption. The deterministic nature of integer arithmetic simplifies execution on hardware, making it particularly well-suited for edge AI and mobile devices. At the extreme end, binary and ternary representations restrict values to just one or two bits, leading to significant reductions in memory footprint and power consumption. However, such aggressive quantization can degrade model accuracy unless complemented by specialized training techniques or architectural adaptations.

Emerging numeric formats seek to balance the trade-off between efficiency and accuracy. TF32, introduced by NVIDIA for Ampere GPUs, modifies FP32 by reducing the mantissa size while maintaining the exponent width, allowing for faster computations with minimal precision loss. Similarly, FP8, which is gaining adoption in AI accelerators, provides an even lower-precision floating-point alternative while retaining a structure that aligns well with machine learning workloads (Micikevicius et al. 2022). Alternative formats such as Posit, Flexpoint, and BF16ALT are also being explored for their potential advantages in numerical stability and hardware adaptability.

The efficiency of numeric encoding is further influenced by how data is stored and accessed in memory. AI accelerators optimize memory hierarchies to maximize the benefits of reduced-precision formats, leveraging specialized hardware such as tensor cores, matrix multiply units (MMUs), and vector processing engines to accelerate lower-precision computations. On these platforms, data alignment, memory tiling, and compression techniques play a crucial role in ensuring that reduced-precision computations deliver tangible performance gains.

As machine learning systems evolve, numeric encoding and storage strategies will continue to adapt to meet the demands of large-scale models and diverse hardware environments. The ongoing development of precision formats tailored for AI workloads highlights the importance of co-designing numerical representations with underlying hardware capabilities, ensuring that machine learning models achieve optimal performance while minimizing computational costs.

10.5.3 Numerical Precision Format Comparison

Table 11.5 compares commonly used numerical precision formats in machine learning, highlighting their trade-offs in storage efficiency, computational speed, and energy consumption. Emerging formats like FP8 and TF32 have been introduced to further optimize performance, particularly on AI accelerators.

Table 10.6: Comparison of numerical precision formats.

Precision Format	Bit-Width	Storage Reduction (vs FP32)	Compute Speed (vs FP32)	Power Consumption	Use Cases
FP32 (Single-Precision Floating Point)	32-bit	Baseline (1×)	Baseline (1×)	High	Training & inference (general-purpose)
FP16 (Half-Precision Floating Point)	16-bit	2× smaller	2× faster on FP16-optimized hardware	Lower	Accelerated training, inference (NVIDIA Tensor Cores, TPUs)
bfloat16 (Brain Floating Point)	16-bit	2× smaller	Similar speed to FP16, better dynamic range	Lower	Training on TPUs, transformer-based models
TF32 (TensorFloat-32)	19-bit	Similar to FP16	Up to 8× faster on NVIDIA Ampere GPUs	Lower	Training on NVIDIA GPUs
FP8 (Floating-Point 8-bit)	8-bit	4× smaller	Faster than INT8 in some cases	Significantly lower	Efficient training/inference (H100, AI accelerators)
INT8 (8-bit Integer)	8-bit	4× smaller	4–8× faster than FP32	Significantly lower	Quantized inference (Edge AI, mobile AI, NPUs)
INT4 (4-bit Integer)	4-bit	8× smaller	Hardware-dependent	Extremely low	Ultra-low-power AI, experimental quantization
Binary/Ternary (1-bit / 2-bit)	1–2-bit	16–32× smaller	Highly hardware-dependent	Lowest	Extreme efficiency (binary/ternary neural networks)

FP16 and bfloat16 formats provide moderate efficiency gains while preserving model accuracy. Many AI accelerators, such as NVIDIA Tensor Cores and TPUs, include dedicated support for FP16 computations, enabling 2× faster matrix operations compared to FP32. BFloat16, in particular, retains the same 8-bit exponent as FP32 but with a reduced 7-bit mantissa, allowing it to maintain a similar dynamic range (~10^{-38} to 10^{38}) while sacrificing precision. In contrast, FP16, with its 5-bit exponent and 10-bit mantissa, has a significantly reduced dynamic range (~10^{-5} to 10^{5}), making it more suitable for inference rather than training. Since BFloat16 preserves the exponent size of FP32, it better handles extreme values encountered during training, whereas FP16 may struggle with underflow or overflow. This makes BFloat16 a more robust alternative for deep learning workloads that require a wide dynamic range.

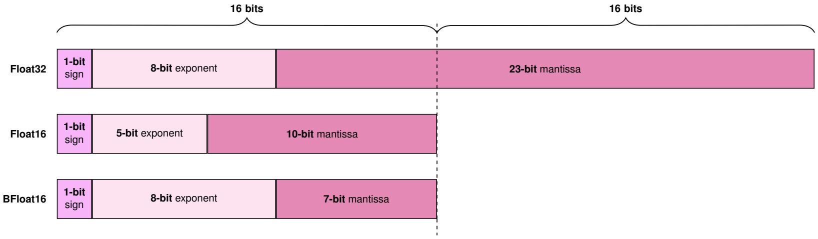


Figure 10.17: Three floating-point formats.

Figure 10.17 highlights these differences, showing how bit-width allocations impact the trade-offs between precision and numerical range.[1]

INT8 precision offers more aggressive efficiency improvements, particularly for inference workloads. Many quantized models use INT8 for inference, reducing storage by $4\times$ while accelerating computation by 4–8× on optimized hardware. INT8 is widely used in mobile and embedded AI, where energy constraints are significant.

Binary and ternary networks represent the extreme end of precision reduction, where weights and activations are constrained to 1-bit (binary) or 2-bit (ternary) values. This results in massive storage and energy savings, but model accuracy often degrades significantly unless specialized architectures are used.

10.5.4 Precision Reduction Trade-offs

Reducing numerical precision in machine learning systems offers substantial gains in efficiency, including lower memory requirements, reduced power consumption, and increased computational throughput. However, these benefits come with trade-offs, as lower-precision representations introduce numerical error and quantization noise, which can affect model accuracy. The extent of this impact depends on multiple factors, including the model architecture, the dataset, and the specific precision format used.

Models exhibit varying levels of tolerance to precision reduction. Large-scale architectures, such as convolutional neural networks and transformer-based models, often retain high accuracy even when using reduced-precision formats

[1] The dynamic range of a floating-point format is determined by its exponent bit-width and bias. FP32 and BFloat16 both use an 8-bit exponent with a bias of 127, resulting in an exponent range of $[-126, 127]$ and an approximate numerical range of 10^{-38} to 10^{38}. FP16, with a 5-bit exponent and a bias of 15, has an exponent range of $[-14, 15]$, leading to a more constrained numerical range of roughly 10^{-5} to 10^{5}. This reduced range in FP16 can lead to numerical instability in training, whereas BFloat16 retains FP32's broader range, making it more suitable for training deep neural networks.

such as bfloat16 or INT8. In contrast, smaller models or those trained on tasks requiring high numerical precision may experience greater degradation in performance. Additionally, not all layers within a neural network respond equally to precision reduction. Certain layers, such as batch normalization and attention mechanisms, may be more sensitive to numerical precision than standard feedforward layers. As a result, techniques such as mixed-precision training, where different layers operate at different levels of precision, can help maintain accuracy while optimizing computational efficiency.

Hardware support is another critical factor in determining the effectiveness of precision reduction. AI accelerators, including GPUs, TPUs, and NPUs, are designed with dedicated low-precision arithmetic units that enable efficient computation using FP16, bfloat16, INT8, and, more recently, FP8. These architectures exploit reduced precision to perform high-throughput matrix operations, improving both speed and energy efficiency. In contrast, general-purpose CPUs often lack specialized hardware for low-precision computations, limiting the potential benefits of numerical precision reduction. The introduction of newer floating-point formats, such as TF32 for NVIDIA GPUs and FP8 for AI accelerators, seeks to optimize the trade-off between precision and efficiency, offering an alternative for hardware that is not explicitly designed for extreme quantization.

In addition to hardware constraints, reducing numerical precision impacts power consumption. Lower-precision arithmetic reduces the number of required memory accesses and simplifies computational operations, leading to lower overall energy use. This is particularly advantageous for energy-constrained environments such as mobile devices and edge AI systems. At the extreme end, ultra-low precision formats, including INT4 and binary/ternary representations, provide substantial reductions in power and memory usage. However, these formats often require specialized architectures to compensate for the accuracy loss associated with such aggressive quantization.

To mitigate accuracy loss associated with reduced precision, various precision reduction strategies can be employed. Ultimately, selecting the appropriate numerical precision for a given machine learning model requires balancing efficiency gains against accuracy constraints. This selection depends on the model's architecture, the computational requirements of the target application, and the underlying hardware's support for low-precision operations. By leveraging advancements in both hardware and software optimization techniques, practitioners can effectively integrate lower-precision numerics into machine learning pipelines, maximizing efficiency while maintaining performance.

10.5.5 Precision Reduction Strategies

Reducing numerical precision is an essential optimization technique for improving the efficiency of machine learning models. By lowering the bit-width of weights and activations, models can reduce memory footprint, improve computational throughput, and decrease power consumption. However, naive precision reduction can introduce quantization errors, leading to accuracy degradation. To address this, different precision reduction strategies have been developed, allowing models to balance efficiency gains while preserving predictive performance.

Precision reduction techniques can be applied at different stages of a model's lifecycle. Post-training quantization reduces precision after training, making it a simple and low-cost approach for optimizing inference. Quantization-aware training incorporates quantization effects into the training process, enabling models to adapt to lower precision and retain higher accuracy. Mixed-precision training leverages hardware support to dynamically assign precision levels to different computations, optimizing execution efficiency without sacrificing accuracy.

10.5.5.1 Post-Training Quantization

Post-training quantization (PTQ) is a widely used technique for optimizing machine learning models by reducing numerical precision after training, improving inference efficiency without requiring additional retraining (Jacob et al. 2018b). By converting model weights and activations from high-precision floating-point formats (e.g., FP32) to lower-precision representations (e.g., INT8 or FP16), PTQ enables smaller model sizes, faster computation, and reduced energy consumption. This makes it a practical choice for deploying models on resource-constrained environments, such as mobile devices, edge AI systems, and cloud inference platforms (H. Wu et al. 2020).

Unlike other quantization techniques that modify the training process, PTQ is applied after training is complete. This means that the model retains its original structure and parameters, but its numerical representation is changed to operate in a more efficient format. The key advantage of PTQ is its low computational cost, as it does not require retraining the model with quantization constraints. However, reducing precision can introduce quantization error, which may lead to accuracy degradation, especially in tasks that rely on fine-grained numerical precision.

PTQ is widely supported in machine learning frameworks such as TensorFlow Lite, ONNX Runtime, and PyTorch's quantization toolkit, making it an accessible and practical approach for optimizing inference workloads. The following sections explore how PTQ works, its benefits and challenges, and techniques for mitigating accuracy loss.

PTQ Functionality. PTQ converts a trained model's weights and activations from high-precision floating-point representations (e.g., FP32) to lower-precision formats (e.g., INT8 or FP16). This process reduces the memory footprint of the model, accelerates inference, and lowers power consumption. However, since lower-precision formats have a smaller numerical range, quantization introduces rounding errors, which can impact model accuracy.

The core mechanism behind PTQ is scaling and mapping high-precision values into a reduced numerical range. A widely used approach is uniform quantization, which maps floating-point values to discrete integer levels using a consistent scaling factor. In uniform quantization, the interval between each quantized value is constant, simplifying implementation and ensuring efficient execution on hardware. The quantized value q is computed as:

$$q = \text{round}\left(\frac{x}{s}\right)$$

where:

- q is the quantized integer representation,
- x is the original floating-point value,
- s is a scaling factor that maps the floating-point range to the available integer range.

For example, in INT8 quantization, the model's floating-point values (typically ranging from $[-r, r]$) are mapped to an integer range of $[-128, 127]$. The scaling factor ensures that the most significant information is retained while reducing precision loss. Once the model has been quantized, inference is performed using integer arithmetic, which is significantly more efficient than floating-point operations on many hardware platforms (A. et al. Gholami 2021). However, due to rounding errors and numerical approximation, quantized models may experience slight accuracy degradation compared to their full-precision counterparts.

Once the model has been quantized, inference is performed using integer arithmetic, which is significantly more efficient than floating-point operations on many hardware platforms. However, due to rounding errors and numerical approximation, quantized models may experience slight accuracy degradation compared to their full-precision counterparts.

In addition to uniform quantization, non-uniform quantization can be employed to preserve accuracy in certain scenarios. Unlike uniform quantization, which uses a consistent scaling factor, non-uniform quantization assigns finer-grained precision to numerical ranges that are more densely populated. This approach can be beneficial for models with weight distributions that concentrate around certain values, as it allows more details to be retained where it matters most. However, non-uniform quantization typically requires more complex calibration and may involve additional computational overhead. While it is not as commonly used as uniform quantization in production environments, non-uniform techniques can be effective for preserving accuracy in models that are particularly sensitive to precision changes.

PTQ is particularly effective for computer vision models, where CNNs often tolerate quantization well. However, models that rely on small numerical differences, such as NLP transformers or speech recognition models, may require additional tuning or alternative quantization techniques, including non-uniform strategies, to retain performance.

Calibration. An important aspect of PTQ is the calibration step[23], which involves selecting the most effective clipping range [α, β] for quantizing model weights and activations. During PTQ, the model's weights and activations are converted to lower-precision formats (e.g., INT8), but the effectiveness of this reduction depends heavily on the chosen quantization range. Without proper calibration, the quantization process may cause significant accuracy degradation, even if the overall precision is reduced. Calibration ensures that

[23] Calibration determines the optimal quantization range for model parameters to minimize information loss during precision reduction.

the chosen range minimizes loss of information and helps preserve the model's performance after precision reduction.

The overall workflow of post-training quantization is illustrated in Figure 10.18. The process begins with a pre-trained model, which serves as the starting point for optimization. To determine an effective quantization range, a calibration dataset, which is a representative subset of training or validation data—is passed through the model. This step allows the calibration process to estimate the numerical distribution of activations and weights, which is then used to define the clipping range for quantization. Following calibration, the quantization step converts the model parameters to a lower-precision format, producing the final quantized model, which is more efficient in terms of memory and computation.

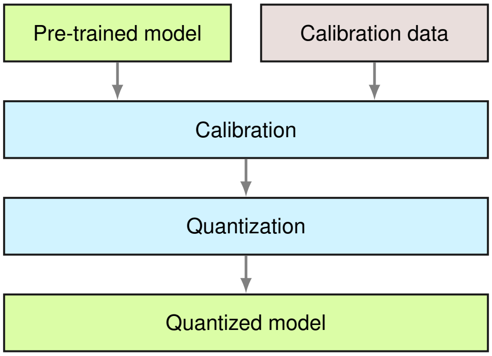


Figure 10.18: Post-Training Quantization Workflow. Calibration uses a pre-trained model and calibration data to determine quantization ranges before applying precision reduction.

For example, consider quantizing activations that originally have a floating-point range between –6 and 6 to 8-bit integers. Simply using the full integer range of –128 to 127 for quantization might not be the most effective approach. Instead, calibration involves passing a representative dataset through the model and observing the actual range of the activations. The observed range can then be used to set a more effective quantization range, reducing information loss.

Calibration Methods. There are several commonly used calibration methods:

- **Max**: This method uses the maximum absolute value seen during calibration as the clipping range. While simple, it is susceptible to outlier data. For example, in the activation distribution shown in Figure 10.19, we see an outlier cluster around 2.1, while the rest of the values are clustered around smaller values. The Max method could lead to an inefficient range if the outliers significantly influence the quantization.
- **Entropy**: This method minimizes information loss between the original floating-point values and the values that could be represented by the quantized format, typically using KL divergence.[24] This is the default calibration method used by TensorRT[25] and works well when trying to preserve the distribution of the original values.
- **Percentile**: This method sets the clipping range to a percentile of the distribution of absolute values seen during calibration. For example, a 99% calibration would clip the top 1% of the largest magnitude values. This

[24] Kullback-Leibler divergence: A measure of how one probability distribution diverges from a second, expected probability distribution.

[25] TensorRT is NVIDIA's high-performance deep learning inference optimizer and runtime engine.

method helps avoid the impact of outliers, which are not representative of the general data distribution.

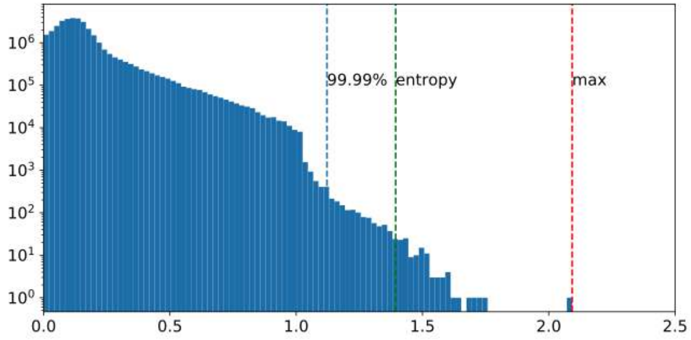


Figure 10.19: Input activations to layer 3 in ResNet50. Source: @H. Wu et al. (2020).

The quality of calibration directly affects the performance of the quantized model. A poor calibration could lead to a model that suffers from significant accuracy loss, while a well-calibrated model can retain much of its original performance after quantization. Importantly, there are two types of calibration ranges to consider:

- **Symmetric Calibration**: The clipping range is symmetric around zero, meaning both the positive and negative ranges are equally scaled.
- **Asymmetric Calibration**: The clipping range is not symmetric, which means the positive and negative ranges may have different scaling factors. This can be useful when the data is not centered around zero.

Choosing the right calibration method and range is critical for maintaining model accuracy while benefiting from the efficiency gains of reduced precision.

Calibration Ranges. A key challenge in post-training quantization is selecting the appropriate calibration range $[\alpha, \beta]$ to map floating-point values into a lower-precision representation. The choice of this range directly affects the quantization error and, consequently, the accuracy of the quantized model. As illustrated in Figure 10.20, there are two primary calibration strategies: symmetric calibration and asymmetric calibration.

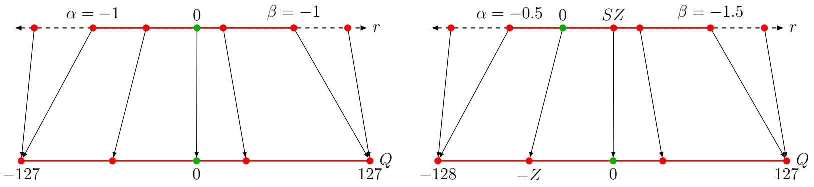


Figure 10.20: Comparison of symmetric and asymmetric calibration methods.

On the left side of Figure 10.20, symmetric calibration is depicted, where the clipping range is centered around zero. The range extends from $\alpha = -1$ to $\beta = 1$, mapping these values to the integer range $[-127, 127]$. This method ensures that positive and negative values are treated equally, preserving zero-centered distributions. A key advantage of symmetric calibration is its simplified implementation, as the same scale factor is applied to both positive and negative values. However, this approach may not be optimal for datasets where the activation distributions are skewed, leading to poor representation of significant portions of the data.

On the right side, asymmetric calibration is shown, where $\alpha = -0.5$ and $\beta = 1.5$. Here, zero is mapped to a shifted quantized value $-Z$, and the range extends asymmetrically. In this case, the quantization scale is adjusted to account for non-zero mean distributions. Asymmetric calibration is particularly useful when activations or weights exhibit skew, ensuring that the full quantized range is effectively utilized. However, it introduces additional computational complexity in determining the optimal offset and scaling factors.

The choice between these calibration strategies depends on the model and dataset characteristics:

- Symmetric calibration is commonly used when weight distributions are centered around zero, which is often the case for well-initialized machine learning models. It simplifies computation and hardware implementation but may not be optimal for all scenarios.
- Asymmetric calibration is useful when the data distribution is skewed, ensuring that the full quantized range is effectively utilized. It can improve accuracy retention but may introduce additional computational complexity in determining the optimal quantization parameters.

Many machine learning frameworks, including TensorRT and PyTorch, support both calibration modes, enabling practitioners to empirically evaluate the best approach. Selecting an appropriate calibration range is important for PTQ, as it directly influences the trade-off between numerical precision and efficiency, ultimately affecting the performance of quantized models.

Granularity. After determining the clipping range, the next step in optimizing quantization involves adjusting the granularity of the clipping range to ensure that the model retains as much accuracy as possible. In CNNs, for instance, the input activations of a layer undergo convolution with multiple convolutional filters, each of which may have a unique range of values. The quantization process, therefore, must account for these differences in range across filters to preserve the model's performance.

As illustrated in Figure 10.21, the range for Filter 1 is significantly smaller than that for Filter 3, demonstrating the variation in the magnitude of values across different filters. The precision with which the clipping range $[\alpha, \beta]$ is determined for the weights becomes a critical factor in effective quantization. This variability in ranges is a key reason why different quantization strategies, based on granularity, are employed.

Several methods are commonly used to determine the granularity of quantization, each with its own trade-offs in terms of accuracy, efficiency, and computational cost.

Layerwise Quantization. In this approach, the clipping range[26] is determined by considering all weights in the convolutional filters of a layer. The same clipping range is applied to all filters within the layer. While this method is

[26] Clipping range: The bounds within which values are allowed before they are adjusted for quantization.

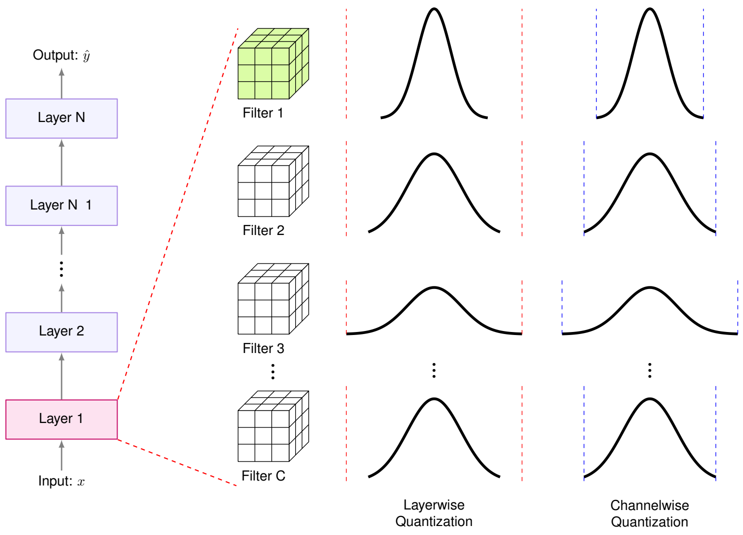


Figure 10.21: Quantization granularity: variable ranges. Source: A. et al. Gholami (2021).

simple to implement, it often leads to suboptimal accuracy due to the wide range of values across different filters. For example, if one convolutional kernel has a narrower range of values than another in the same layer, the quantization resolution of the narrower range may be compromised, resulting in a loss of information.

Groupwise Quantization. Groupwise quantization divides the convolutional filters into groups and calculates a shared clipping range for each group. This method can be beneficial when the distribution of values within a layer is highly variable. For example, the Q-BERT model (Shen et al. 2019) applied this technique when quantizing Transformer models (M. X. Chen et al. 2018), particularly for the fully-connected attention layers. While groupwise quantization offers better accuracy than layerwise quantization, it incurs additional computational cost due to the need to account for multiple scaling factors.

Channelwise Quantization. Channelwise quantization assigns a dedicated clipping range and scaling factor to each convolutional filter. This approach ensures a higher resolution in quantization, as each channel is quantized independently. Channelwise quantization is widely used in practice, as it often yields better accuracy compared to the previous methods. By allowing each filter to have its own clipping range, this method ensures that the quantization process is tailored to the specific characteristics of each filter.

Sub-channelwise Quantization. This method takes the concept of channelwise quantization a step further by subdividing each convolutional filter into smaller groups, each with its own clipping range. Although this method can provide very fine-grained control over quantization, it introduces significant computational overhead as multiple scaling factors must be managed for each group within a filter. As a result, sub-channelwise quantization is generally only used in scenarios where maximum precision is required, despite the increased computational cost.

Among these methods, channelwise quantization is the current standard for quantizing convolutional filters. It strikes a balance between the accuracy gains from finer granularity and the computational efficiency needed for practical deployment. Adjusting the clipping range for each individual kernel provides significant improvements in model accuracy with minimal overhead, making it the most widely adopted approach in machine learning applications.

Weights vs. Activations. Weight Quantization involves converting the continuous, high-precision weights of a model into lower-precision values, such as converting 32-bit floating-point (Float32) weights to 8-bit integer (INT8) weights. As illustrated in Figure 10.22, weight quantization occurs in the second step (red squares) during the multiplication of inputs. This process significantly reduces the model size, decreasing both the memory required to store the model and the computational resources needed for inference. For example, a weight matrix in a neural network layer with Float32 weights like $[0.215, -1.432, 0.902, \ldots]$ might be mapped to INT8 values such as $[27, -183, 115, \ldots]$, leading to a substantial reduction in memory usage.

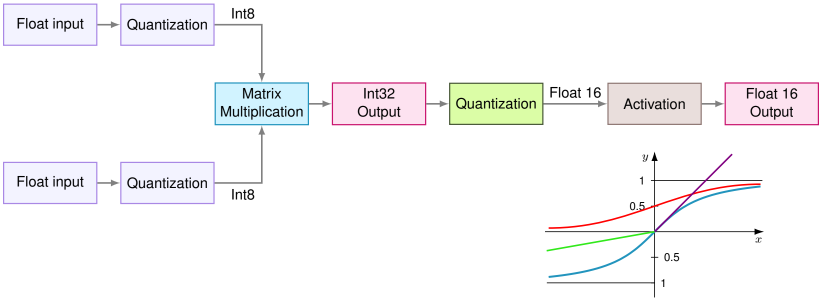


Figure 10.22: Weight and activation quantization. Source: HarvardX.

Activation Quantization refers to the process of quantizing the activation values, or outputs of the layers, during model inference. This quantization can reduce the computational resources required during inference, particularly when targeting hardware optimized for integer arithmetic. It introduces challenges related to maintaining model accuracy, as the precision of intermediate computations is reduced. For instance, in a CNN, the activation maps (or feature maps) produced by convolutional layers, originally represented in Float32, may be quantized to INT8 during inference. This can significantly accelerate computation on hardware capable of efficiently processing lower-precision integers.

Recent advancements have explored Activation-aware Weight Quantization (AWQ) for the compression and acceleration of large language models (LLMs). This approach focuses on protecting only a small fraction of the most salient weights, approximately 1%, by observing the activations rather than the weights themselves. This method has been shown to improve model efficiency while preserving accuracy, as discussed in (Ji Lin, Tang, et al. 2023).

Static vs. Dynamic Quantization. After determining the type and granularity of the clipping range, practitioners must decide when the clipping ranges are calculated in their quantization algorithms. Two primary approaches exist for quantizing activations: static quantization and dynamic quantization.

Static Quantization is the more commonly used approach. In static quantization, the clipping range is pre-calculated and remains fixed during inference. This method does not introduce any additional computational overhead during runtime, which makes it efficient in terms of computational resources. However, the fixed range can lead to lower accuracy compared to dynamic quantization. A typical implementation of static quantization involves running a series of calibration inputs to compute the typical range of activations, as discussed in works like (Jacob et al. 2018b) and (Yao et al. 2021).

In contrast, Dynamic Quantization dynamically calculates the range for each activation map during runtime. This approach allows the quantization process to adjust in real time based on the input, potentially yielding higher accuracy since the range is specifically calculated for each input activation. However, dynamic quantization incurs higher computational overhead because the range must be recalculated at each step. Although this often results in higher accuracy, the real-time computations can be expensive, particularly when deployed at scale.

The following table, Table 10.7, summarizes the characteristics of post-training quantization, quantization-aware training, and dynamic quantization, providing an overview of their respective strengths, limitations, and trade-offs. These methods are widely deployed across machine learning systems of varying scales, and understanding their pros and cons is crucial for selecting the appropriate approach for a given application.

Table 10.7: Comparison of post-training quantization, quantization-aware training, and dynamic quantization.

Aspect	Post Training Quantization	Quantization-Aware Training	Dynamic Quantization
Pros			
Simplicity	✓		
Accuracy Preservation		✓	✓
Adaptability			✓
Optimized Performance		✓	Potentially
Cons			
Accuracy Degradation	✓		Potentially
Computational Overhead		✓	✓
Implementation Complexity		✓	✓
Tradeoffs			
Speed vs. Accuracy	✓		
Accuracy vs. Cost		✓	
Adaptability vs. Overhead			✓

PTQ Advantages. One of the key advantages of PTQ is its low computational cost, as it does not require retraining the model. This makes it an attractive option for the rapid deployment of trained models, particularly when retraining is computationally expensive or infeasible. Since PTQ only modifies the numerical representation of weights and activations, the underlying model architecture remains unchanged, allowing it to be applied to a wide range of pre-trained models without modification.

PTQ also provides substantial memory and storage savings by reducing the bit-width of model parameters. For instance, converting a model from FP32 to INT8 results in a $4\times$ reduction in storage size, making it feasible to deploy larger models on resource-constrained devices such as mobile phones, edge AI hardware, and embedded systems. These reductions in memory footprint also lead to lower bandwidth requirements when transferring models across networked systems.

In terms of computational efficiency, PTQ allows inference to be performed using integer arithmetic, which is inherently faster than floating-point operations on many hardware platforms. AI accelerators such as TPUs and Neural Processing Units (NPUs) are optimized for lower-precision computations, enabling higher throughput and reduced power consumption when executing quantized models. This makes PTQ particularly useful for applications requiring real-time inference, such as object detection in autonomous systems or speech recognition on mobile devices.

PTQ Challenges and Limitations. Despite its advantages, PTQ introduces quantization errors due to rounding effects when mapping floating-point values to discrete lower-precision representations. While some models remain robust to these changes, others may experience notable accuracy degradation, especially in tasks that rely on small numerical differences.

The extent of accuracy loss depends on both the model architecture and the task domain. CNNs for image classification are generally tolerant to PTQ, often maintaining near-original accuracy even with aggressive INT8 quantization. Transformer-based models used in natural language processing (NLP) and speech recognition tend to be more sensitive, as these architectures rely on the precision of numerical relationships in attention mechanisms.

To mitigate accuracy loss, calibration techniques such as KL divergence-based scaling or per-channel quantization are commonly applied to fine-tune the scaling factor and minimize information loss. Some frameworks, including TensorFlow Lite and PyTorch, provide automated quantization tools with built-in calibration methods to improve accuracy retention.

Another limitation of PTQ is that not all hardware supports efficient integer arithmetic. While GPUs, TPUs, and specialized edge AI chips often include dedicated support for INT8 inference, general-purpose CPUs may lack the optimized instructions for low-precision execution, resulting in suboptimal performance improvements.

Additionally, PTQ is not always suitable for training purposes. Since PTQ applies quantization after training, models that require further fine-tuning or adaptation may benefit more from alternative approaches, such as quantization-aware training (which we will discuss next), to ensure that precision constraints are adequately considered during the learning process.

Post-training quantization remains one of the most practical and widely used techniques for improving inference efficiency. It provides substantial memory and computational savings with minimal overhead, making it an ideal choice for deploying machine learning models in resource-constrained environments.

However, the success of PTQ depends on model architecture, task sensitivity, and hardware compatibility. In scenarios where accuracy degradation is unacceptable, alternative quantization strategies, such as quantization-aware training, may be required.

10.5.5.2 Quantization-Aware Training

While PTQ offers a fast, computationally inexpensive approach for optimizing inference efficiency, it has inherent limitations; applying quantization after training does not consider the impacts of reduced numerical precision on model behavior. This oversight can result in noticeable accuracy degradation, particularly for models that rely on fine-grained numerical precision, such as transformers used in NLP and speech recognition systems (Nagel et al. 2021a).

QAT addresses this limitation by integrating quantization constraints directly into the training process. Instead of reducing precision after training, QAT simulates low-precision arithmetic during forward passes, allowing the model to learn how to be more robust to quantization effects. This ensures that the model's accuracy is better maintained once deployed with low-precision computations (Jacob et al. 2018b).

As illustrated in Figure 10.23, QAT involves first applying quantization to a pre-trained model, followed by retraining or fine-tuning using training data. This process allows the model to adapt to low-precision numerical constraints, mitigating accuracy degradation.

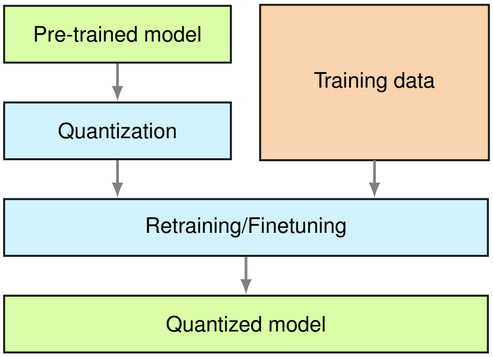


Figure 10.23: Quantization-aware training process.

In many cases, QAT can also build off PTQ, as shown in Figure 10.24. Instead of starting from a full-precision model, PTQ is first applied to produce an initial quantized model, leveraging calibration data to determine appropriate quantization parameters. This PTQ model then serves as the starting point for QAT, where additional fine-tuning with training data helps the model better adapt to low-precision constraints. This hybrid approach benefits from the efficiency of PTQ while reducing the accuracy degradation typically associated with post-training quantization alone.

Training Mathematics. During forward propagation, weights and activations are quantized and dequantized to mimic reduced precision. This process is

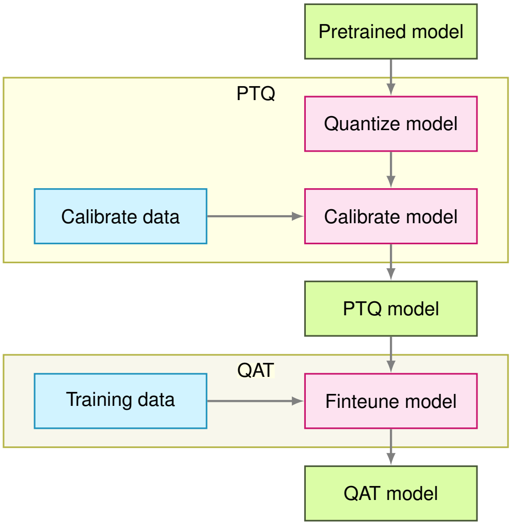


Figure 10.24: Quantization-aware training process after PTQ.

typically represented as:

$$q = \text{round}\left(\frac{x}{s}\right) \times s$$

where q represents the simulated quantized value, x denotes the full-precision weight or activation, and s is the scaling factor mapping floating-point values to lower-precision integers.

Although the forward pass utilizes quantized values, gradient calculations during backpropagation remain in full precision. This is achieved using the Straight-Through Estimator (STE)[27], which approximates the gradient of the quantized function by treating the rounding operation as if it had a derivative of one. This approach prevents the gradient from being obstructed due to the non-differentiable nature of the quantization operation, thereby allowing effective model training (Y. Bengio, Léonard, and Courville 2013a).

Integrating quantization effects during training enables the model to learn an optimal distribution of weights and activations that minimizes the impact of numerical precision loss. The resulting model, when deployed using true low-precision arithmetic (e.g., INT8 inference), maintains significantly higher accuracy than one that is quantized post hoc (Krishnamoorthi 2018).

QAT Advantages. A primary advantage of QAT is its ability to maintain model accuracy, even under low-precision inference conditions. Incorporating quantization during training helps the model to compensate for precision loss, reducing the impact of rounding errors and numerical instability. This is critical for quantization-sensitive models commonly used in NLP, speech recognition, and high-resolution computer vision (A. et al. Gholami 2021).

Another major benefit is that QAT permits low-precision inference on hardware accelerators without significant accuracy degradation. AI processors such as TPUs, NPUs, and specialized edge devices include dedicated hardware for integer operations, permitting INT8 models to run much faster and with lower

[27] Straight-Through Estimator (STE): A method used in neural network training to backpropagate through non-differentiable functions.

energy consumption compared to FP32 models. Training with quantization effects in mind ensures that the final model can fully leverage these hardware optimizations (H. Wu et al. 2020).

QAT Challenges and Trade-offs. Despite its benefits, QAT introduces additional computational overhead during training. Simulated quantization at every forward pass slows down training relative to full-precision methods. The process adds complexity to the training schedule, making QAT less practical for very large-scale models where the additional training time might be prohibitive.

Moreover, QAT introduces extra hyperparameters and design considerations, such as choosing appropriate quantization schemes and scaling factors. Unlike PTQ, which applies quantization after training, QAT requires careful tuning of the training dynamics to ensure that the model suitably adapts to low-precision constraints (Gong et al. 2019).

Table 10.8 summarizes the key trade-offs of QAT compared to PTQ:

Table 10.8: Comparison of QAT and PTQ.

Aspect	QAT (Quantization-Aware Training)	PTQ (Post-Training Quantization)
Accuracy Retention	Minimizes accuracy loss from quantization	May suffer from accuracy degradation
Inference Efficiency	Optimized for low-precision hardware (e.g., INT8 on TPUs)	Optimized but may require calibration
Training Complexity	Requires retraining with quantization constraints	No retraining required
Training Time	Slower due to simulated quantization in forward pass	Faster, as quantization is applied post hoc
Deployment Readiness	Best for models sensitive to quantization errors	Fastest way to optimize models for inference

Integrating quantization into the training process preserves model accuracy more effectively than post-training quantization, although it requires additional training resources and time.

10.5.5.3 PTQ and QAT Strategies

PTQ and QAT are supported across modern machine learning frameworks, facilitating efficient deployment of machine learning models on low-precision hardware. Although PTQ is simpler to implement since it does not require retraining, QAT embeds quantization into the training pipeline, leading to better accuracy retention. Each framework offers specialized tools that allow these methods to be applied effectively while balancing computational trade-offs.

TensorFlow implements PTQ using `tf.lite.TFLiteConverter`, which converts model weights and activations to lower-precision formats (e.g., INT8) post-training. Since PTQ circumvents retraining, calibration techniques such as per-channel quantization and KL-divergence scaling can be applied to minimize accuracy loss. TensorFlow also supports QAT through `tf.keras.quantization.quantize_model()`, which leverages simulated quantization operations inserted into the computation graph. This allows models to learn weight distributions more robust to reduced precision, thereby improving accuracy when deployed with INT8 inference.

In PyTorch, PTQ is performed using `torch.quantization.convert()`, which transforms a pre-trained model into a quantized version optimized for inference. PyTorch supports both dynamic and static quantization, enabling trade-offs between accuracy and efficiency. QAT in PyTorch is facilitated using `torch.quantization.prepare_qat()`, which introduces fake quantization layers during training to simulate low-precision arithmetic while maintaining full-precision gradients. This approach helps the model adapt to quantization constraints without incurring substantial accuracy loss.

ONNX Runtime supports PTQ through `onnxruntime.quantization`, which includes both static and dynamic quantization modes. While static quantization relies on calibration data to determine optimal scaling factors for weights and activations, dynamic quantization applies quantization only during inference, offering flexibility for real-time applications. For QAT, ONNX Runtime provides `onnxruntime.training.QuantizationMode.QAT`, allowing models to be trained with simulated quantization prior to export for INT8 inference.

Although PTQ offers a straightforward and computationally inexpensive means to optimize models, it may lead to accuracy degradation—especially for sensitivity-critical architectures. QAT, despite its higher training cost, delivers models that better preserve accuracy when deployed under low-precision computations.

10.5.5.4 PTQ vs. QAT

Quantization plays a critical role in optimizing machine learning models for deployment on low-precision hardware, enabling smaller model sizes, faster inference, and reduced power consumption. The choice between PTQ and QAT depends on the trade-offs between accuracy, computational cost, and deployment constraints.

PTQ is the preferred approach when retraining is infeasible or unnecessary. It is computationally inexpensive, requiring only a conversion step after training, making it an efficient way to optimize models for inference. However, its effectiveness varies across model architectures—CNNs for image classification often tolerate PTQ well, while NLP and speech models may experience accuracy degradation due to their reliance on precise numerical representations.

QAT, in contrast, is necessary when high accuracy retention is critical. By integrating quantization effects during training, QAT allows models to adapt to lower-precision arithmetic, reducing quantization errors. While this results in higher accuracy in low-precision inference, it also requires additional training time and computational resources, making it less practical for cases where fast model deployment is a priority (Jacob et al. 2018c).

Ultimately, the decision between PTQ and QAT depends on the specific requirements of the machine learning system. If rapid deployment and minimal computational overhead are the primary concerns, PTQ provides a quick and effective solution. If accuracy is a critical factor and the model is sensitive to quantization errors, QAT offers a more robust but computationally expensive alternative. In many real-world applications, a hybrid approach that starts with PTQ and selectively applies QAT for accuracy-critical models provides the best balance between efficiency and performance.

10.5.6 Extreme Precision Reduction

Extreme precision reduction techniques, such as binarization and ternarization, are designed to dramatically reduce the bit-width of weights and activations in a neural network. By representing values with just one or two bits (for binary and ternary representations, respectively), these techniques achieve substantial reductions in memory usage and computational requirements, making them particularly attractive for hardware-efficient deployment in resource-constrained environments (Courbariaux, Bengio, and David 2016).

10.5.6.1 Binarization

Binarization involves reducing weights and activations to just two values, typically -1 and +1, or 0 and 1, depending on the specific method. The primary advantage of binarization lies in its ability to drastically reduce the size of a model, allowing it to fit into a very small memory footprint. This reduction also accelerates inference, especially when deployed on specialized hardware such as binary neural networks (Rastegari et al. 2016). However, binarization introduces significant challenges, primarily in terms of model accuracy. When weights and activations are constrained to only two values, the expressiveness of the model is greatly reduced, which can lead to a loss in accuracy, particularly in tasks requiring high precision, such as image recognition or natural language processing (Hubara et al. 2018).

Moreover, the process of binarization introduces non-differentiable operations, which complicates the optimization process. To address this issue, techniques such as the STE are employed to approximate gradients, allowing for effective backpropagation despite the non-differentiability of the quantization operation (Y. Bengio, Léonard, and Courville 2013b). The use of STE ensures that the network can still learn and adjust during training, even with the extreme precision reduction. While these challenges are non-trivial, the potential benefits of binarized models in ultra-low-power environments, such as edge devices and IoT sensors, make binarization an exciting area of research.

10.5.6.2 Ternarization

Ternarization extends binarization by allowing three possible values for weights and activations—typically -1, 0, and +1. While ternarization still represents a significant reduction in precision, it offers a slight improvement in model accuracy over binarization, as the additional value (0) provides more flexibility in capturing the underlying patterns (Zhu et al. 2017). This additional precision comes at the cost of increased complexity, both in terms of computation and the required training methods. Similar to binarization, ternarization is often implemented using techniques that approximate gradients, such as the hard thresholding method or QAT, which integrate quantization effects into the training process to mitigate the accuracy loss (J. Choi et al. 2018).

The advantages of ternarization over binarization are most noticeable when dealing with highly sparse data[28]. In some cases, ternarization can introduce more sparsity into the model by mapping a large portion of weights to zero. However, managing this sparsity effectively requires careful implementation to

[28] Data with a high proportion of zero values, often found in large datasets.

avoid the overhead that comes with storing sparse matrices (F. Li et al. 2016). Additionally, while ternarization improves accuracy compared to binarization, it still represents a severe trade-off in terms of the model's ability to capture intricate relationships between inputs and outputs. The challenge, therefore, lies in finding the right balance between the memory and computational savings offered by ternarization and the accuracy loss incurred by reducing the precision.

10.5.6.3 Computation Challenges and Limitations

What makes binarization and ternarization particularly interesting is their potential to enable ultra-low-power machine learning. These extreme precision reduction methods offer a way to make machine learning models more feasible for deployment on hardware with strict resource constraints, such as embedded systems and mobile devices. However, the challenge remains in how to maintain the performance of these models despite such drastic reductions in precision. Binarized and ternarized models require specialized hardware that is capable of efficiently handling binary or ternary operations. Many traditional processors are not optimized for this type of computation, which means that realizing the full potential of these methods often requires custom hardware accelerators (Umuroglu et al. 2017).

Another challenge is the loss of accuracy that typically accompanies the extreme precision reduction inherent in binarization and ternarization. These methods are best suited for tasks where high levels of precision are not critical, or where the model can be trained to adjust to the precision constraints through techniques like QAT. Despite these challenges, the ability to drastically reduce the size of a model while maintaining acceptable levels of accuracy makes binarization and ternarization attractive for certain use cases, particularly in edge AI and resource-constrained environments (Jacob et al. 2018c).

The future of these techniques lies in advancing both the algorithms and hardware that support them. As more specialized hardware is developed for low-precision operations, and as techniques for compensating for precision loss during training improve, binarization and ternarization will likely play a significant role in making AI models more efficient, scalable, and energy-efficient.

10.5.7 Quantization vs. Model Representation

Thus far, we explored various quantization techniques, including PTQ, QAT, and extreme precision reduction methods like binarization and ternarization. These techniques aim to reduce the memory footprint and computational demands of machine learning models, making them suitable for deployment in environments with strict resource constraints, such as edge devices or mobile platforms.

While quantization offers significant reductions in model size and computational requirements, it often requires careful management of the trade-offs between model efficiency and accuracy. When comparing quantization to other model representation techniques, including pruning, knowledge distillation, and NAS, several key differences and synergies emerge.

Pruning focuses on reducing the number of parameters in a model by removing unimportant or redundant weights. While quantization reduces the precision of weights and activations, pruning reduces their sheer number. The two techniques can complement each other: pruning can be applied first to reduce the number of weights, which then makes the quantization process more effective by working with a smaller set of parameters. However, pruning does not necessarily reduce precision, so it may not achieve the same level of computational savings as quantization.

Knowledge distillation reduces model size by transferring knowledge from a large, high-precision model (teacher) to a smaller, more efficient model (student). While quantization focuses on precision reduction within a given model, distillation works by transferring learned behavior into a more compact model. The advantage of distillation is that it can help mitigate accuracy loss, which is often a concern with extreme precision reduction. When combined with quantization, distillation can help ensure that the smaller, quantized model retains much of the accuracy of the original, larger model.

NAS automates the design of neural network architectures to identify the most efficient model for a given task. NAS focuses on optimizing the structure of the model itself, whereas quantization operates on the numerical representation of the model's weights and activations. The two approaches can be complementary, as NAS can lead to model architectures that are inherently more suited for low-precision operations, thus making quantization more effective. In this sense, NAS can be seen as a precursor to quantization, as it optimizes the architecture for the constraints of low-precision environments.

As shown in Figure 10.25, different compression strategies such as pruning, quantization, and singular value decomposition (SVD) exhibit varying trade-offs between model size and accuracy loss. While pruning combined with quantization (red circles) achieves high compression ratios with minimal accuracy loss, quantization alone (yellow squares) also provides a reasonable balance. In contrast, SVD (green diamonds) requires a larger model size to maintain accuracy, illustrating how different techniques can impact compression effectiveness.

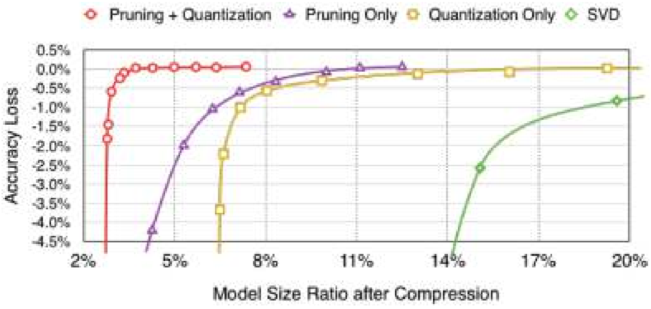


Figure 10.25: Accuracy vs. compression rate under different compression methods. Source: Han, Mao, and Dally (2015).

In summary, quantization differs from pruning, knowledge distillation, and NAS in that it specifically focuses on reducing the numerical precision of weights and activations. While quantization alone can provide significant computational benefits, its effectiveness can be amplified when combined with the complementary techniques of pruning, distillation, and NAS. These methods, each targeting a different aspect of model efficiency, work together to create more compact, faster, and energy-efficient models, enabling better performance in constrained environments. By understanding the strengths and limitations of these methods, practitioners can choose the most suitable combination to meet the specific needs of their application and deployment hardware.

10.6 Architectural Efficiency Optimization

Architectural efficiency is the process of optimizing the machine learning model structures with an explicit focus on the computational resources available during deployment. Unlike other optimization methods, such as pruning and knowledge distillation, which are applied after model training and are agnostic to the hardware on which the model will run, architectural efficiency requires proactive consideration of the target hardware from the beginning. This approach ensures that models are designed to effectively utilize the specific capabilities of the deployment platform, whether it be a mobile device, embedded system, or specialized AI hardware.

10.6.1 Hardware-Aware Design

The incorporation of hardware constraints, including memory bandwidth, processing power, and energy consumption, into model design enables the creation of architectures that are both accurate and computationally efficient. This approach leads to improved performance and reduced resource usage during both training and inference.

The focus of this section is on the techniques that can be employed to achieve architectural efficiency, including exploiting sparsity, model factorization, dynamic computation, and hardware-aware design. These techniques allow for the development of models that are optimized for the constraints of specific hardware environments, ensuring that they can operate efficiently and meet the performance requirements of real-world applications.

10.6.1.1 Efficient Design Principles

Designing machine learning models for hardware efficiency requires structuring architectures to account for computational cost, memory usage, inference latency, and power consumption, all while maintaining strong predictive performance. Unlike post-training optimizations, which attempt to recover efficiency after training, hardware-aware model design proactively integrates hardware considerations from the outset. This ensures that models are computationally efficient and deployable across diverse hardware environments with minimal adaptation.

A key aspect of hardware-aware design is leveraging the strengths of specific hardware platforms (e.g., GPUs, TPUs, mobile or edge devices) to maximize parallelism, optimize memory hierarchies, and minimize latency through hardware-optimized operations. As summarized in Table 10.9, hardware-aware model design can be categorized into several principles, each addressing a fundamental aspect of computational and system constraints.

Table 10.9: Taxonomy of hardware-aware model design principles.

Principle	Goal	Example Networks
Scaling Optimization	Adjust model depth, width, and resolution to balance efficiency and hardware constraints.	EfficientNet, RegNet
Computation Reduction	Minimize redundant operations to reduce computational cost, utilizing hardware-specific optimizations (e.g., using depthwise separable convolutions on mobile chips).	MobileNet, ResNeXt
Memory Optimization	Ensure efficient memory usage by reducing activation and parameter storage requirements, leveraging hardware-specific memory hierarchies (e.g., local and global memory in GPUs).	DenseNet, SqueezeNet
Hardware-Aware Design	Optimize architectures for specific hardware constraints (e.g., low power, parallelism, high throughput).	TPU-optimized models, MobileNet

Hardware-aware model design principles focus on creating efficient architectures that align with specific hardware capabilities and constraints. Scaling optimization ensures models are appropriately sized for available resources, preventing inefficient parameterization while maintaining performance. This allows models to effectively utilize hardware without exceeding its limitations.

Computation reduction techniques eliminate redundant operations that consume excessive resources. For example, mobile CPUs and GPUs can leverage parallelism through depthwise separable convolutions, reducing computational overhead while preserving model effectiveness. These optimizations are specifically tailored to take advantage of hardware-specific features.

Memory optimization plays a crucial role by considering hardware-specific memory hierarchies, including cache and on-chip memory. This principle ensures efficient data movement and maximizes throughput by aligning memory access patterns with the underlying hardware architecture. Proper memory management reduces bottlenecks and improves overall system performance.

Hardware-aware design aligns architectural decisions directly with platform capabilities. By optimizing model structure for specific hardware, this approach ensures maximum execution efficiency and minimal power consumption. The result is a model that not only performs well but also operates efficiently within hardware constraints.

These principles work together synergistically, enabling the creation of models that balance accuracy with computational efficiency. By considering hardware constraints during the design phase, models achieve better performance while maintaining lower resource usage, particularly when deployed on platforms with limited capabilities.

10.6.1.2 Scaling Optimization

Scaling a model's architecture involves balancing accuracy with computational cost, and optimizing it to align with the capabilities of the target hardware. Each component of a model, whether its depth, width, or input resolution, impacts resource consumption. In hardware-aware design, these dimensions should not only be optimized for accuracy but also for efficiency in memory usage, processing power, and energy consumption, especially when the model is deployed on specific hardware like GPUs, TPUs, or edge devices.

From a hardware-aware perspective, it is crucial to consider how different hardware platforms, such as GPUs, TPUs, or edge devices, interact with scaling dimensions. For instance, deeper models can capture more complex representations, but excessive depth can lead to increased inference latency, longer training times, and higher memory consumption—issues that are particularly problematic on resource-constrained platforms. Similarly, increasing the width of the model to process more parallel information may be beneficial for GPUs and TPUs with high parallelism, but it requires careful management of memory usage. In contrast, increasing the input resolution can provide finer details for tasks like image classification, but it exponentially increases computational costs, potentially overloading hardware memory or causing power inefficiencies on edge devices.

Mathematically, the total FLOPs for a convolutional model can be approximated as:

$$\text{FLOPs} \propto d \cdot w^2 \cdot r^2,$$

where d is depth, w is width, and r is the input resolution. Increasing all three dimensions without considering the hardware limitations can result in suboptimal performance, especially on devices with limited computational power or memory bandwidth.

For efficient model scaling, it's essential to manage these parameters in a balanced way, ensuring that the model remains within the limits of the hardware while maximizing performance. This is where compound scaling comes into play. Instead of adjusting depth, width, and resolution independently, compound scaling balances all three dimensions together by applying fixed ratios (α, β, γ) relative to a base model:

$$d = \alpha^\phi d_0, \quad w = \beta^\phi w_0, \quad r = \gamma^\phi r_0$$

Here, ϕ is a scaling coefficient, and α, β, and γ are scaling factors determined based on hardware constraints and empirical data. This approach ensures that models grow in a way that optimizes hardware resource usage, keeping them efficient while improving accuracy.

For example, EfficientNet, which employs compound scaling, demonstrates how carefully balancing depth, width, and resolution results in models that are both computationally efficient and high-performing. Compound scaling reduces computational cost while preserving accuracy, making it a key consideration for hardware-aware model design. This approach is particularly beneficial when deploying models on GPUs or TPUs, where parallelism can be fully leveraged, but memory and power usage need to be carefully managed.

This principle applies not only to convolutional models but also to other architectures like transformers. For instance, in transformer models, adjusting the number of attention heads or layers can have similar resource usage implications, and hardware-aware scaling can ensure that the computational cost is minimized while maintaining strong performance.

Beyond convolutional models, this principle of scaling optimization can be generalized to other architectures, such as transformers. In these architectures, adjusting the number of layers, attention heads, or embedding dimensions can have a similar impact on computational efficiency. Hardware-aware scaling has become a central consideration in optimizing model performance for various computational constraints, particularly when working with large models or resource-constrained devices.

10.6.1.3 Computation Reduction

Reducing redundant operations is a critical strategy for improving the efficiency of machine learning models, especially when considering deployment on resource-constrained hardware. Traditional machine learning architectures, particularly convolutional neural networks, often rely on dense operations, including standard convolutions, which apply computations uniformly across all spatial locations and channels. However, these operations introduce unnecessary computation, especially when many of the channels or activations do not contribute meaningfully to the final prediction. This can lead to excessive computational load and memory consumption, which are significant constraints on hardware with limited processing power or memory bandwidth, like mobile or embedded devices.

To address this issue, modern architectures leverage factorized computations, which decompose complex operations into simpler components. This enables models to achieve the same representational power while reducing the computational overhead, making them more efficient for deployment on specific hardware platforms. One widely adopted method for computation reduction is depthwise separable convolutions, introduced in the MobileNet architecture. Depthwise separable convolutions break a standard convolution operation into two distinct steps:

1. Depthwise convolution applies a separate convolutional filter to each input channel independently, ensuring that computations for each channel are treated separately.
2. Pointwise convolution (a 1×1 convolution) then mixes the outputs across channels, effectively combining the results into the final feature representation.

This factorization reduces the number of operations compared to the standard convolutional approach, where a single filter processes all input channels simultaneously. The reduction in operations is particularly beneficial for hardware accelerators, as it reduces the number of calculations that need to be performed and the amount of memory bandwidth required. The computational complexity of a standard convolution with an input size of $h \times w$, C_{in} input channels, and C_{out} output channels can be expressed as:

$$\mathcal{O}(hwC_{\text{in}}C_{\text{out}}k^2)$$

where k is the kernel size. This equation shows that the computational cost scales with both the spatial dimensions and the number of channels, making it computationally expensive. However, for depthwise separable convolutions, the complexity reduces to:

$$\mathcal{O}(hwC_{\text{in}}k^2) + \mathcal{O}(hwC_{\text{in}}C_{\text{out}})$$

Here, the first term depends only on C_{in}, the number of input channels, and the second term eliminates the k^2 factor from the channel-mixing operation.

The result is a $5\times$ to $10\times$ reduction in FLOPs (floating-point operations), which directly reduces the computational burden and improves model efficiency, particularly for hardware with limited resources, such as mobile devices or edge processors.

Beyond depthwise separable convolutions, other architectures employ additional factorization techniques to further reduce computation. For example, Grouped convolutions, used in the ResNeXt architecture, split feature maps into separate groups, each of which is processed independently before being merged. This approach increases computational efficiency while maintaining strong accuracy by reducing redundant operations. Another example is Bottleneck layers, used in architectures like ResNet. These layers employ 1×1 convolutions to reduce the dimensionality of feature maps before applying larger convolutions, which reduces the computational complexity of deeper networks, where most of the computational cost lies.

These computation reduction techniques are highly effective in optimizing models for specific hardware, particularly for real-time applications in mobile, edge computing, and embedded systems. By reducing the number of computations required, models can achieve high performance while consuming fewer resources, which is critical for ensuring low inference latency and minimal energy usage.

In hardware-aware model design, such as when deploying on GPUs, TPUs, or other specialized accelerators, these techniques can significantly reduce computational load and memory footprint. By reducing the complexity of operations, the hardware can process the data more efficiently, allowing for faster execution and lower power consumption. Additionally, these techniques can be combined with other optimizations, such as sparsity, to maximize hardware utilization and achieve better overall performance.

10.6.1.4 Memory Optimization

Memory optimization is a fundamental aspect of model efficiency, especially when deploying machine learning models on resource-constrained hardware, such as mobile devices, embedded systems, and edge AI platforms. Inference-based models require memory to store activations, intermediate feature maps, and parameters. If these memory demands exceed the hardware's available resources, the model can experience performance bottlenecks, including increased inference latency and power inefficiencies due to frequent memory accesses. Efficient memory management is crucial to minimize these issues while maintaining accuracy and performance.

To address these challenges, modern architectures employ various memory-efficient strategies that reduce unnecessary storage while keeping the model's performance intact. Hardware-aware memory optimization techniques are particularly important when considering deployment on accelerators such as GPUs, TPUs, or edge AI chips. These strategies ensure that models are computationally tractable and energy-efficient, particularly when operating under strict power and memory constraints.

One effective technique for memory optimization is feature reuse[29], a strategy employed in DenseNet. In traditional convolutional networks, each layer

[29] Feature Reuse: A strategy where networks reuse the computed features from previous layers to reduce computational redundancy.

typically computes a new set of feature maps, increasing the model's memory footprint. However, DenseNet reduces the need for redundant activations by reusing feature maps from previous layers and selectively applying transformations. This method reduces the total number of feature maps that need to be stored, which in turn lowers the memory requirements without sacrificing accuracy. In a standard convolutional network with L layers, if each layer generates k new feature maps, the total number of feature maps grows linearly:

$$\mathcal{O}(Lk)$$

In contrast, DenseNet reuses feature maps from earlier layers, reducing the number of feature maps stored. This leads to improved parameter efficiency and a reduced memory footprint, which is essential for hardware with limited memory resources.

Another useful technique is activation checkpointing, which is especially beneficial during training. In a typical neural network, backpropagation requires storing all forward activations for the backward pass. This can lead to a significant memory overhead, especially for large models. Activation checkpointing reduces memory consumption by only storing a subset of activations and recomputing the remaining ones when needed. If an architecture requires storing A_{total} activations, the standard backpropagation method requires the full storage:

$$\mathcal{O}(A_{\text{total}})$$

With activation checkpointing, however, only a fraction of activations is stored, and the remaining ones are recomputed on-the-fly, reducing storage requirements to:

$$\mathcal{O}\left(\sqrt{A_{\text{total}}}\right)$$

This technique can significantly reduce peak memory consumption, making it particularly useful for training large models on hardware with limited memory.

Parameter reduction is another essential technique, particularly for models that use large filters. For instance, SqueezeNet uses a novel architecture where it applies 1×1 convolutions to reduce the number of input channels before applying standard convolutions.[30] By first reducing the number of channels with 1×1 convolutions, SqueezeNet reduces the model size significantly without compromising the model's expressive power. The number of parameters in a standard convolutional layer is:

$$\mathcal{O}(C_{\text{in}} C_{\text{out}} k^2)$$

By reducing C_{in} using 1×1 convolutions, SqueezeNet reduces the number of parameters, achieving a 50x reduction in model size compared to AlexNet while maintaining similar performance. This method is particularly valuable for edge devices that have strict memory and storage constraints.

These memory-efficient techniques, including feature reuse, activation checkpointing, and parameter reduction, are key components of hardware-aware model design. By minimizing memory usage and efficiently managing storage, these techniques allow machine learning models to fit within the memory limits of modern accelerators, such as GPUs, TPUs, and edge devices. These strategies

[30] SqueezeNet achieves similar accuracy to AlexNet while being 50 times smaller.

also lead to lower power consumption by reducing the frequency of memory accesses, which is particularly beneficial for devices with limited battery life.

In hardware-aware design, memory optimization is not just about reducing memory usage but also about optimizing how memory is accessed. Specialized accelerators like TPUs and GPUs can take advantage of memory hierarchies, caching, and high bandwidth memory to efficiently handle sparse or reduced-memory representations. By incorporating these memory-efficient strategies, models can operate with minimal overhead, enabling faster inference and more efficient power consumption.

10.6.2 Dynamic Computation and Adaptation

Dynamic computation refers to the ability of a machine learning model to adapt its computational load based on the complexity of the input. Rather than applying a fixed amount of computation to every input, dynamic computation allows models to allocate computational resources more effectively, depending on the task's requirements. This is especially crucial for applications where computational efficiency, real-time processing, and energy conservation are vital, such as in mobile devices, embedded systems, and autonomous vehicles.

In traditional machine learning models, every input is processed using the same network architecture, irrespective of its complexity. For example, an image classification model might apply the full depth of a neural network to classify both a simple and a complex image, even though the simple image could be classified with fewer operations. This uniform processing results in wasted computational resources, unnecessary power consumption, and increased processing times—all of which are particularly problematic in real-time and resource-constrained systems.

Dynamic computation addresses these inefficiencies by allowing models to adjust the computational load based on the input's complexity. For simpler inputs, the model might skip certain layers or operations, reducing computational costs. On the other hand, for more complex inputs, it may opt to process additional layers or operations to ensure accuracy is maintained. This adaptive approach not only optimizes computational efficiency but also reduces energy consumption, minimizes latency, and preserves high predictive performance.

Dynamic computation is essential for efficient resource use on hardware with limited capabilities. Adjusting the computational load dynamically based on input complexity enables models to significantly enhance efficiency and overall performance without sacrificing accuracy.

10.6.2.1 Dynamic Schemes

Dynamic schemes enable models to selectively reduce computation when inputs are simple, preserving resources while maintaining predictive performance. The approaches discussed below, beginning with early exit architectures, illustrate how to implement this adaptive strategy effectively.

Early Exit Architectures. Early exit architectures allow a model to make predictions at intermediate points in the network rather than completing the full forward pass for every input. This approach is particularly effective for real-time applications and energy-efficient inference, as it enables selective computation based on the complexity of individual inputs (Teerapittayanon, McDanel, and Kung 2017).

The core mechanism in early exit architectures involves multiple exit points embedded within the network. Simpler inputs, which can be classified with high confidence early in the model, exit at an intermediate layer, reducing unnecessary computations. Conversely, more complex inputs continue processing through deeper layers to ensure accuracy.

A well-known example is BranchyNet, which introduces multiple exit points throughout the network. For each input, the model evaluates intermediate predictions using confidence thresholds. If the prediction confidence exceeds a predefined threshold at an exit point, the model terminates further computations and outputs the result. Otherwise, it continues processing until the final layer (Teerapittayanon, McDanel, and Kung 2017). This approach minimizes inference time without compromising performance on challenging inputs.

Another example is multi-exit vision transformers, which extend early exits to transformer-based architectures. These models use lightweight classifiers at various transformer layers, allowing predictions to be generated early when possible (Scardapane, Wang, and Panella 2020). This technique significantly reduces inference time while maintaining robust performance for complex samples.

Early exit models are particularly advantageous for resource-constrained devices, such as mobile processors and edge accelerators. By dynamically adjusting computational effort, these architectures reduce power consumption and processing latency, making them ideal for real-time decision-making (B. Hu, Zhang, and Fu 2021).

When deployed on hardware accelerators such as GPUs and TPUs, early exit architectures can be further optimized by exploiting parallelism. For instance, different exit paths can be evaluated concurrently, thereby improving throughput while preserving the benefits of adaptive computation (Yu, Li, and Wang 2023). This approach is illustrated in Figure 10.26, where each transformer layer is followed by a classifier and an optional early exit mechanism based on confidence estimation or latency-to-accuracy trade-offs (LTE). At each stage, the system may choose to exit early if sufficient confidence is achieved, or continue processing through deeper layers, enabling dynamic allocation of computational resources.

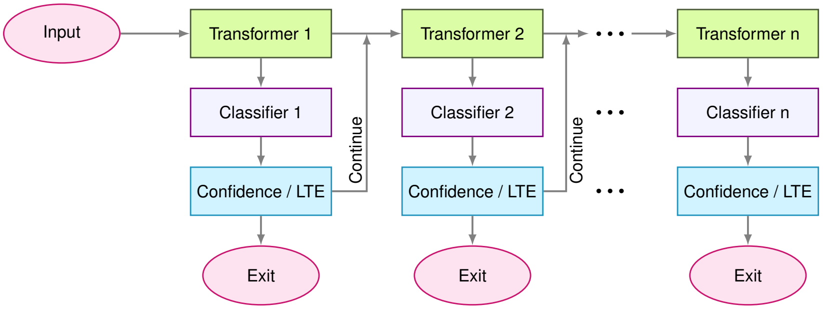


Figure 10.26: Example early exit BERT architecture, where a confidence score per Transformer layer is used to determine whether to exit early. Source: Xin et al. (2021).

Conditional Computation. Conditional computation refers to the ability of a neural network to decide which parts of the model to activate based on the input, thereby reducing unnecessary computation. This approach can be highly beneficial in resource-constrained environments, such as mobile devices or real-time systems, where reducing the number of operations directly translates to lower computational cost, power consumption, and inference latency (E. Bengio et al. 2015).

In contrast to Early Exit Architectures, where the decision to exit early is typically made once a threshold confidence level is met, conditional computation works by dynamically selecting which layers, units, or paths in the network should be computed based on the characteristics of the input. This can be achieved through mechanisms such as gating functions or dynamic routing, which essentially "turn off" parts of the network that are not needed for a particular input, allowing the model to focus computational resources where they are most required.

One example of conditional computation is SkipNet, which uses a gating mechanism to skip layers in a CNN when the input is deemed simple enough. The gating mechanism uses a lightweight classifier to predict if the layer should be skipped. This prediction is made based on the input, and the model adjusts the number of layers used during inference accordingly (X. Wang et al. 2018). If the gating function determines that the input is simple, certain layers are bypassed, resulting in faster inference. However, for more complex inputs, the model uses the full depth of the network to achieve the necessary accuracy.

Another example is Dynamic Routing Networks, such as in the Capsule Networks (CapsNets), where routing mechanisms dynamically choose the path that activations take through the network. In these networks, the decision-making process involves selecting specific pathways for information flow based on the input's complexity, which can significantly reduce the number of operations and computations required (Sabour, Frosst, and Hinton 2017). This mechanism introduces adaptability by leveraging different routing strategies, providing computational efficiency while preserving the quality of predictions.

These conditional computation strategies have significant advantages in real-world applications where computational resources are limited. For example, in autonomous driving, the system must process a variety of inputs (e.g., pedestrians, traffic signs, road lanes) with varying complexity. In cases where the input is straightforward, a simpler, less computationally demanding path can be taken, whereas more complex scenarios (such as detecting obstacles or performing detailed scene understanding) will require full use of the model's capacity. Conditional computation ensures that the system adapts its computation based on the real-time complexity of the input, leading to improved speed and efficiency (W. Huang, Chen, and Zhang 2023).

Gate-Based Computation. Gate-based conditional computation introduces learned gating mechanisms that dynamically control which parts of a neural network are activated based on input complexity. Unlike static architectures that process all inputs with the same computational effort, this approach enables dynamic activation of sub-networks or layers by learning decision boundaries during training (Shazeer, Mirhoseini, Maziarz, and others 2017).

Gating mechanisms are typically implemented using binary or continuous gating functions, wherein a lightweight control module (often called a router or gating network) predicts whether a particular layer or path should be executed. This decision-making occurs dynamically at inference time, allowing the model to allocate computational resources adaptively.

A well-known example of this paradigm is the Dynamic Filter Network (DFN), which applies input-dependent filtering by selecting different convolutional kernels at runtime. DFN reduces unnecessary computation by avoiding uniform filter application across inputs, tailoring its computations based on input complexity (Xu Jia et al. 2016).

Another widely adopted strategy is the Mixture of Experts (MoE) framework. In this architecture, a gating network selects a subset of specialized expert subnetworks to process each input (Shazeer, Mirhoseini, Maziarz, and others 2017). This allows only a small portion of the total model to be active for any given input, significantly improving computational efficiency without sacrificing model capacity. A notable instantiation of this idea is Google's Switch Transformer, which extends the transformer architecture with expert-based conditional computation (Fedus, Zoph, and Shazeer 2021).

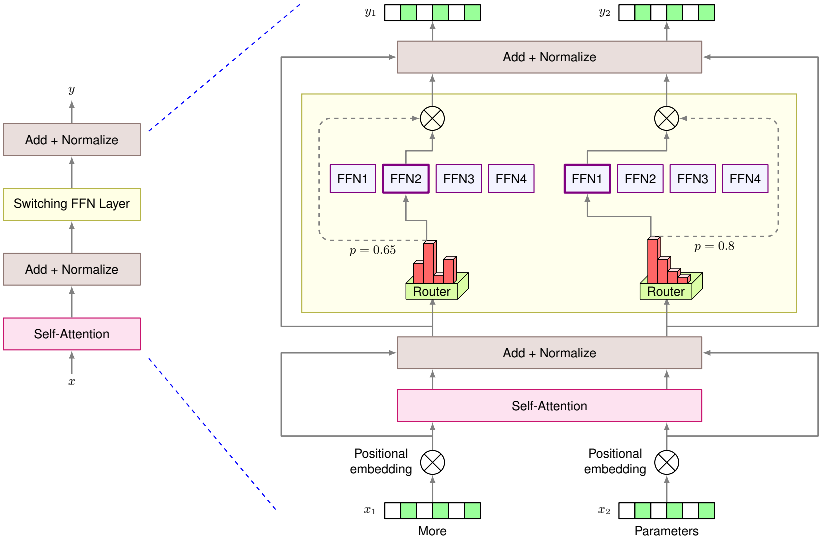


Figure 10.27: A Switch Transformer block is an example of Mixture of Experts (MoE) architecture and an architecture that dynamic routes token computation to subnetworks. Source (Fedus, Zoph, and Shazeer 2021).

As shown in Figure 10.27, the Switch Transformer replaces the traditional feedforward layer with a Switching FFN Layer. For each token, a lightweight router selects a single expert from a pool of feedforward networks. The router outputs a probability distribution over available experts, and the highest-probability expert is activated per token. This design enables large models to scale parameter count without proportionally increasing inference cost.

Gate-based conditional computation is particularly effective for multi-task and transfer learning settings, where inputs may benefit from specialized processing pathways. By enabling fine-grained control over model execution, such mechanisms allow for adaptive specialization across tasks while maintaining efficiency.

However, these benefits come at the cost of increased architectural complexity. The routing and gating operations themselves introduce additional overhead, both in terms of latency and memory access. Efficient deployment, particularly on hardware accelerators such as GPUs, TPUs, or edge devices, requires careful attention to the scheduling and batching of expert activations (Lepikhin et al. 2020).

Adaptive Inference. Adaptive inference refers to a model's ability to dynamically adjust its computational effort during inference based on input complexity. Unlike earlier approaches that rely on predefined exit points or discrete layer skipping, adaptive inference continuously modulates computational depth and resource allocation based on real-time confidence and task complexity (Yang et al. 2020).

This flexibility allows models to make on-the-fly decisions about how much computation is required, balancing efficiency and accuracy without rigid thresholds. Instead of committing to a fixed computational path, adaptive inference enables models to dynamically allocate layers, operations, or specialized computations based on intermediate assessments of the input (Yang et al. 2020).

One example of adaptive inference is Fast Neural Networks (FNNs), which adjust the number of active layers based on real-time complexity estimation. If an input is deemed straightforward, only a subset of layers is activated, reducing inference time. However, if early layers produce low-confidence outputs, additional layers are engaged to refine the prediction (Jian Wu, Cheng, and Zhang 2019).

A related approach is dynamic layer scaling, where models progressively increase computational depth based on uncertainty estimates. This technique is particularly useful for fine-grained classification tasks, where some inputs require only coarse-grained processing while others need deeper feature extraction (Contro et al. 2021).

Adaptive inference is particularly effective in latency-sensitive applications where resource constraints fluctuate dynamically. For instance, in autonomous systems, tasks such as lane detection may require minimal computation, while multi-object tracking in dense environments demands additional processing power. By adjusting computational effort in real-time, adaptive inference ensures that models operate within strict timing constraints without unnecessary resource consumption.

On hardware accelerators such as GPUs and TPUs, adaptive inference leverages parallel processing capabilities by distributing workloads dynamically. This adaptability maximizes throughput while minimizing energy expenditure, making it ideal for real-time, power-sensitive applications.

10.6.2.2 Computation Challenges and Limitations

Dynamic computation introduces flexibility and efficiency by allowing models to adjust their computational workload based on input complexity. However, this adaptability comes with several challenges that must be addressed to make dynamic computation practical and scalable. These challenges arise in training, inference efficiency, hardware execution, generalization, and evaluation, each presenting unique difficulties that impact model design and deployment.

Training and Optimization Difficulties. Unlike standard neural networks, which follow a fixed computational path for every input, dynamic computation requires additional control mechanisms, such as gating networks, confidence estimators, or expert selection strategies. These mechanisms determine which parts of the model should be activated or skipped, adding complexity to the training process. One major difficulty is that many of these decisions are discrete, meaning they cannot be optimized using standard backpropagation. Instead, models often rely on techniques like reinforcement learning or continuous approximations, but these approaches introduce additional computational costs and can slow down convergence.

Training dynamic models also presents instability because different inputs follow different paths, leading to inconsistent gradient updates across training examples. This variability can make optimization less efficient, requiring careful regularization strategies[31] to maintain smooth learning dynamics. Additionally, dynamic models introduce new hyperparameters, such as gating thresholds or confidence scores for early exits. Selecting appropriate values for these parameters is crucial to ensuring the model effectively balances accuracy and efficiency, but it significantly increases the complexity of the training process.

Overhead and Latency Variability. Although dynamic computation reduces unnecessary operations, the process of determining which computations to perform introduces additional overhead. Before executing inference, the model must first decide which layers, paths, or subnetworks to activate. This decision-making process, often implemented through lightweight gating networks, adds computational cost and can partially offset the savings gained by skipping computations. While these overheads are usually small, they become significant in resource-constrained environments where every operation matters.

An even greater challenge is the variability in inference time. In static models, inference follows a fixed sequence of operations, leading to predictable execution times. In contrast, dynamic models exhibit variable processing times depending on input complexity. For applications with strict real-time constraints, such as autonomous driving or robotics, this unpredictability can be problematic. A model that processes some inputs in milliseconds but others in significantly longer time frames may fail to meet strict latency requirements, limiting its practical deployment.

Hardware Execution Inefficiencies. Modern hardware accelerators, such as GPUs and TPUs, are optimized for uniform, parallel computation patterns. These accelerators achieve maximum efficiency by executing identical operations across large batches of data simultaneously. However, dynamic computation introduces conditional branching, which can disrupt this parallel execution model. When different inputs follow different computational paths, some processing units may remain idle while others are active, leading to suboptimal hardware utilization.

This divergent execution pattern creates significant challenges for hardware efficiency. For example, in a GPU where multiple threads process data in

[31] Regularization: A method used in neural networks to prevent overfitting in models by adding a cost term to the loss function.

parallel, conditional branches cause thread divergence, where some threads must wait while others complete their operations. Similarly, TPUs are designed for large matrix operations and achieve peak performance when all processing units are fully utilized. Dynamic computation can prevent these accelerators from maintaining high throughput, potentially reducing the cost-effectiveness of deployment at scale.

The impact is particularly pronounced in scenarios requiring real-time processing or high-throughput inference. When hardware resources are not fully utilized, the theoretical computational benefits of dynamic computation may not translate into practical performance gains. This inefficiency becomes more significant in large-scale deployments where maximizing hardware utilization is crucial for managing operational costs and maintaining service-level agreements.

Memory access patterns also become less predictable in dynamic models. Standard machine learning models process data in a structured manner, optimizing for efficient memory access. In contrast, dynamic models require frequent branching, leading to irregular memory access and increased latency. Optimizing these models for hardware execution requires specialized scheduling strategies and compiler optimizations to mitigate these inefficiencies, but such solutions add complexity to deployment.

Generalization and Robustness. Because dynamic computation allows different inputs to take different paths through the model, there is a risk that certain data distributions receive less computation than necessary. If the gating functions are not carefully designed, the model may learn to consistently allocate fewer resources to specific types of inputs, leading to biased predictions. This issue is particularly concerning in safety-critical applications, where failing to allocate enough computation to rare but important inputs can result in catastrophic failures.

Another concern is overfitting to training-time computational paths. If a model is trained with a certain distribution of computational choices, it may struggle to generalize to new inputs where different paths should be taken. Ensuring that a dynamic model remains adaptable to unseen data requires additional robustness mechanisms, such as entropy-based regularization or uncertainty-driven gating, but these introduce additional training complexities.

Dynamic computation also creates new vulnerabilities to adversarial attacks. In standard models, an attacker might attempt to modify an input in a way that alters the final prediction. In dynamic models, an attacker could manipulate the gating mechanisms themselves, forcing the model to choose an incorrect or suboptimal computational path. Defending against such attacks requires additional security measures that further complicate model design and deployment.

Evaluation and Benchmarking. Most machine learning benchmarks assume a fixed computational budget, making it difficult to evaluate the performance of dynamic models. Traditional metrics such as FLOPs or latency do not fully capture the adaptive nature of these models, where computation varies based on input complexity. As a result, standard benchmarks fail to reflect the true trade-offs between accuracy and efficiency in dynamic architectures.

Another issue is reproducibility. Because dynamic models make input-dependent decisions, running the same model on different hardware or under slightly different conditions can lead to variations in execution paths. This variability complicates fair comparisons between models and requires new evaluation methodologies to accurately assess the benefits of dynamic computation. Without standardized benchmarks that account for adaptive scaling, it remains challenging to measure and compare dynamic models against their static counterparts in a meaningful way.

Despite these challenges, dynamic computation remains a promising direction for optimizing efficiency in machine learning. Addressing these limitations requires more robust training techniques, hardware-aware execution strategies, and improved evaluation frameworks that properly account for dynamic scaling. As machine learning continues to scale and computational constraints become more pressing, solving these challenges will be key to unlocking the full potential of dynamic computation.

10.6.3 Sparsity Exploitation

Sparsity in machine learning refers to the condition where a substantial portion of the elements within a tensor, such as weight matrices or activation tensors, are zero or nearly zero. More formally, for a tensor $T \in \mathbb{R}^{m \times n}$ (or higher dimensions), the sparsity S can be expressed as:

$$S = \frac{\|\mathbf{1}_{\{T_{ij}=0\}}\|_0}{m \times n}$$

where $\mathbf{1}_{\{T_{ij}=0\}}$ is an indicator function that yields 1 if $T_{ij} = 0$ and 0 otherwise, and $\|\cdot\|_0$ represents the L0 norm, which counts the number of non-zero elements.

Due to the nature of floating-point representations, we often extend this definition to include elements that are close to zero. This leads to:

$$S_\epsilon = \frac{\|\mathbf{1}_{\{|T_{ij}|<\epsilon\}}\|_0}{m \times n}$$

where ϵ is a small threshold value.

Sparsity can emerge naturally during training, often as a result of regularization techniques, or be deliberately introduced through methods like pruning, where elements below a specific threshold are forced to zero. Effectively exploiting sparsity leads to significant computational efficiency, memory savings, and reduced power consumption, which are particularly valuable when deploying models on devices with limited resources, such as mobile phones, embedded systems, and edge devices.

10.6.3.1 Sparsity Types

Sparsity in neural networks can be broadly classified into two types: unstructured sparsity and structured sparsity.

Unstructured sparsity occurs when individual weights are set to zero without any specific pattern. This type of sparsity can be achieved through techniques like pruning, where weights that are considered less important (often based on magnitude or other criteria) are removed. While unstructured sparsity is highly flexible and can be applied to any part of the network, it can be less efficient on hardware since it lacks a predictable structure. In practice, exploiting unstructured sparsity requires specialized hardware or software optimizations to make the most of it.

In contrast, structured sparsity involves removing entire components of the network, such as filters, neurons, or channels, in a more systematic manner. By eliminating entire parts of the network, structured sparsity is more efficient on hardware accelerators like GPUs or TPUs, which can leverage this structure for faster computations. Structured sparsity is often used when there is a need for predictability and efficiency in computational resources, as it enables the hardware to fully exploit regular patterns in the network.

10.6.3.2 Sparsity Exploitation Techniques

To exploit sparsity effectively in neural networks, several key techniques can be used. These techniques reduce the memory and computational burden of the model while preserving its performance. However, the successful application of these techniques often depends on the availability of specialized hardware support to fully leverage sparsity (Hoefler, Alistarh, Ben-Nun, Dryden, and Peste 2021).

Pruning is one of the most widely used methods to introduce sparsity in neural networks. Pruning involves the removal of less important weights or entire components from the network, effectively reducing the number of parameters. This process can be applied in either an unstructured or structured manner. In unstructured pruning, individual weights are removed based on their importance, while structured pruning involves removing entire filters, channels, or layers (Han et al. 2015). While pruning is highly effective for reducing model size and computation, it requires specialized algorithms and hardware support to fully optimize sparse networks.

Another technique for exploiting sparsity is sparse matrix operations. In sparse matrices, many elements are zero, and these matrices can be stored and processed efficiently, allowing for matrix multiplications with fewer computations. This can be achieved by skipping over the zero elements during the calculation, which significantly reduces the number of arithmetic operations. Specialized hardware, such as GPUs and TPUs, can accelerate these sparse operations by supporting the efficient processing of matrices that contain many zero values (Baraglia and Konno 2019).

For example, consider multiplying a dense 4×4 matrix with a dense vector. In a typical dense implementation, 16 multiplications would be required. However, with sparse-aware implementation, the model only computes the 6 nonzero multiplications, skipping over the zeros. This leads to significant computational savings, especially as the size of the matrix grows.

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 5 & 0 \\ 0 & 0 & 0 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 2x_1 + x_4 \\ 3x_2 \\ 4x_1 + 5x_3 \\ 6x_4 \end{bmatrix}$$

A third important technique for exploiting sparsity is low-rank approximation. In this approach, large, dense weight matrices are approximated by smaller, lower-rank matrices that capture the most important information while discarding redundant components. This reduces both the storage requirements and computational cost. For instance, a weight matrix of size 1000×1000 with one million parameters can be factorized into two smaller matrices, say U (size 1000×50) and V (size 50×1000), which results in only 100,000 parameters—much fewer than the original one million. This smaller representation retains the key features of the original matrix while significantly reducing the computational burden (Emily Denton 2014).

Low-rank approximations, such as Singular Value Decomposition, are commonly used to compress weight matrices in neural networks. These approximations are widely applied in recommendation systems and natural language processing models to reduce computational complexity and memory usage without a significant loss in performance (Joulin et al. 2017).

In addition to these core methods, other techniques like sparsity-aware training can also help models to learn sparse representations during training. For instance, using sparse gradient descent, where the training algorithm updates only non-zero elements, can help the model operate with fewer active parameters. While pruning and low-rank approximations directly reduce parameters or factorize weight matrices, sparsity-aware training helps maintain efficient models throughout the training process (C. Liu et al. 2018).

10.6.3.3 Sparsity Hardware Support

Sparsity is a technique for reducing computational cost, memory usage, and power consumption. However, the full potential of sparsity can only be realized when it is supported by hardware designed to efficiently process sparse data and operations. While general-purpose processors like CPUs are capable of handling basic computations, they are not optimized for the specialized tasks that sparse matrix operations require (Han, Mao, and Dally 2016). This limitation can prevent the potential efficiency gains of sparse networks from being fully realized.

To overcome this limitation, hardware accelerators such as GPUs, TPUs, and FPGAs are increasingly used to accelerate sparse network computations. These accelerators are designed with specialized architectures that can exploit sparsity to improve computation speed, memory efficiency, and power usage. In particular, GPUs, TPUs, and FPGAs can handle large-scale matrix operations more efficiently by skipping over zero elements in sparse matrices, leading to significant reductions in both computational cost and memory bandwidth usage (A. Gholami et al. 2021).

The role of hardware support for sparsity is integral to the broader goal of model optimization. While sparsity techniques, including pruning and low-rank approximation, serve to simplify and compress neural networks, hardware accelerators ensure that these optimizations lead to actual performance gains during training and inference. Therefore, hardware considerations are a critical component of model optimization, as specialized accelerators are necessary to efficiently process sparse data and achieve the desired reductions in both computation time and resource consumption.

Furthermore, sparse operations can also be well mapped onto hardware via software. For example, MegaBlocks (Gale et al. 2022) reformulates sparse Mixture of Experts training into block-sparse operations and develops GPU specific kernels to efficiently handle the sparsity of these computations on hardware and maintain high accelerator utilization.

10.6.3.4 Common Structured Sparsity Patterns

Various sparsity formats have been developed, each with unique structural characteristics and implications. Two of the most prominent are block sparse matrices and N:M sparsity patterns. Block sparse matrices generally have isolated blocks of zero and non-zero dense submatricies such that a matrix operation on the large sparse matrix can be easily re-expressed as a smaller (overall arithmetic-wise) number of dense operations on submatricies. This sparsity allows more efficient storage of the dense submatricies while maintaining shape compatibility for operations like matrix or vector products. For example, Figure 10.28 shows how NVIDIA's cuSPARSE library supports sparse block matrix operations and storage. Several other works, such as Monarch matrices (Dao et al. 2022), have extended on this block-sparsity to strike an improved balance between matrix expressivity and compute/memory efficiency.

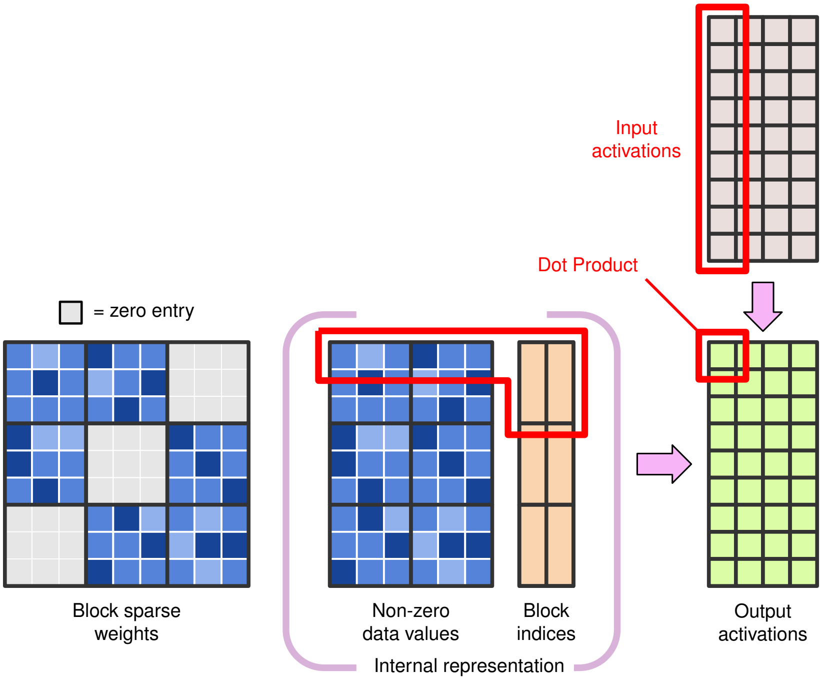


Figure 10.28: Block sparse matrix multiplication implemented in cuSPARSE, showing compressed internal representation while maintaining compatibility with dense matrices via block indices.

Similarly, the N:M sparsity pattern is a structured sparsity format where, in every set of M consecutive elements (e.g., weights or activations), exactly N are non-zero, and the other two are zero (Zhou et al. 2021). This deterministic pattern facilitates efficient hardware acceleration, as it allows for predictable memory access patterns and optimized computations. By enforcing this structure, models can achieve a balance between sparsity-induced efficiency gains and maintaining sufficient capacity for learning complex representations. Figure 10.29 below shows a comparison between accelerating dense versus 2:4 sparsity matrix multiplication, a common sparsity pattern used in model training. Later works like STEP (Lu et al. 2023) have examined learning more general N:M sparsity masks for accelerating deep learning inference under the same principles.

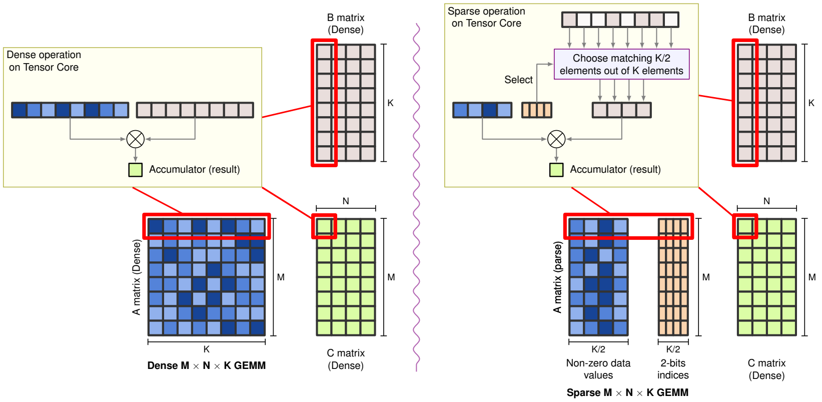


Figure 10.29: Illustration of 2:4 (sparse) matrix multiplication on NVIDIA GPUs. Source PyTorch Blog

GPUs and Sparse Operations. Graphics Processing Units (GPUs) are widely recognized for their ability to perform highly parallel computations, making them ideal for handling the large-scale matrix operations that are common in machine learning. Modern GPUs, such as NVIDIA's Ampere architecture, include specialized Sparse Tensor Cores that accelerate sparse matrix multiplications. These tensor cores are designed to recognize and skip over zero elements in sparse matrices, thereby reducing the number of operations required (Abdelkhalik et al. 2022). This is particularly advantageous for structured pruning techniques, where entire filters, channels, or layers are pruned, resulting in a significant reduction in the amount of computation. By skipping over the zero values, GPUs can speed up matrix multiplications by a factor of two or more, resulting in lower processing times and reduced power consumption for sparse networks.

Furthermore, GPUs leverage their parallel architecture to handle multiple operations simultaneously. This parallelism is especially beneficial for sparse operations, as it allows the hardware to exploit the inherent sparsity in the data more efficiently. However, the full benefit of sparse operations on GPUs requires that the sparsity is structured in a way that aligns with the underlying hardware architecture, making structured pruning more advantageous for optimization (Hoefler, Alistarh, Ben-Nun, Dryden, and Peste 2021).

TPUs and Sparse Optimization. TPUs, developed by Google, are custom-built hardware accelerators specifically designed to handle tensor computations at a much higher efficiency than traditional processors. TPUs, such as TPU v4, have built-in support for sparse weight matrices, which is particularly beneficial for models like transformers, including BERT and GPT, that rely on large-scale matrix multiplications (Jouppi et al. 2021a). TPUs optimize sparse weight matrices by reducing the computational load associated with zero elements, enabling faster processing and improved energy efficiency.

The efficiency of TPUs comes from their ability to perform operations at high throughput and low latency, thanks to their custom-designed matrix multiply units. These units are able to accelerate sparse matrix operations by directly processing the non-zero elements, making them well-suited for models that incorporate significant sparsity, whether through pruning or low-rank approximations. As the demand for larger models increases, TPUs continue to play a critical role in maintaining performance while minimizing the energy and computational cost associated with dense computations.

FPGAs and Sparse Computations. Field-Programmable Gate Arrays (FPGAs) are another important class of hardware accelerators for sparse networks. Unlike GPUs and TPUs, FPGAs are highly customizable, offering flexibility in their design to optimize specific computational tasks. This makes them particularly suitable for sparse operations that require fine-grained control over hardware execution. FPGAs can be programmed to perform sparse matrix-vector multiplications and other sparse matrix operations with minimal overhead, delivering high performance for models that use unstructured pruning or require custom sparse patterns.

One of the main advantages of FPGAs in sparse networks is their ability to be tailored for specific applications, which allows for optimizations that general-purpose hardware cannot achieve. For instance, an FPGA can be designed to skip over zero elements in a matrix by customizing the data path and memory management, providing significant savings in both computation and memory usage. FPGAs also allow for low-latency execution, making them well-suited for real-time applications that require efficient processing of sparse data streams.

Memory and Energy Optimization. One of the key challenges in sparse networks is managing memory bandwidth, as matrix operations often require significant memory access. Sparse networks offer a solution by reducing the number of elements that need to be accessed, thus minimizing memory traffic. Hardware accelerators are optimized for these sparse matrices, utilizing specialized memory access patterns that skip zero values, reducing the total amount of memory bandwidth used (Baraglia and Konno 2019).

For example, GPUs and TPUs are designed to minimize memory access latency by taking advantage of their high memory bandwidth. By accessing only non-zero elements, these accelerators ensure that memory is used more efficiently. The memory hierarchies in these devices are also optimized for sparse computations, allowing for faster data retrieval and reduced power consumption.

The reduction in the number of computations and memory accesses directly translates into energy savings. Sparse operations require fewer arithmetic operations and fewer memory fetches, leading to a decrease in the energy consumption required for both training and inference. This energy efficiency is particularly important for applications that run on edge devices, where power constraints are critical. Hardware accelerators like TPUs and GPUs are optimized to handle these operations efficiently, making sparse networks not only faster but also more energy-efficient (Y. et al. Cheng 2022).

Future: Hardware and Sparse Networks. As hardware continues to evolve, we can expect more innovations tailored specifically for sparse networks. Future hardware accelerators may offer deeper integration with sparsity-aware training and optimization algorithms, allowing even greater reductions in computational and memory costs. Emerging fields like neuromorphic computing, inspired by the brain's structure, may provide new avenues for processing sparse networks in energy-efficient ways (M. et al. Davies 2021). These advancements promise to further enhance the efficiency and scalability of machine learning models, particularly in applications that require real-time processing and run on power-constrained devices.

10.6.3.5 Sparsity Challenges and Limitations

While exploiting sparsity offers significant advantages in reducing computational cost and memory usage, several challenges and limitations must be considered for the effective implementation of sparse networks. Table 10.10 summarizes some of the challenges and limitations associated with sparsity optimizations.

Table 10.10: Challenges and limitations of sparsity optimization for architectural efficiency.

Challenge	Description	Impact
Unstructured Sparsity Optimization	Irregular sparse patterns make it difficult to exploit sparsity on hardware.	Limited hardware acceleration and reduced computational savings.
Algorithmic Complexity	Sophisticated pruning and sparse matrix operations require complex algorithms.	High computational overhead and algorithmic complexity for large models.
Hardware Support	Hardware accelerators are optimized for structured sparsity, making unstructured sparsity harder to optimize.	Suboptimal hardware utilization and lower performance for unstructured sparsity.
Accuracy Trade-off	Aggressive sparsity may degrade model accuracy if not carefully balanced.	Potential loss in performance, requiring careful tuning and validation.
Energy Efficiency	Overhead from sparse matrix storage and management can offset the energy savings from reduced computation.	Power consumption may not improve if the overhead surpasses savings from sparse computations.
Limited Applicability	Sparsity may not benefit all models or tasks, especially in domains requiring dense representations.	Not all models or hardware benefit equally from sparsity.

One of the main challenges of sparsity is the optimization of unstructured sparsity. In unstructured pruning, individual weights are removed based on their importance, leading to an irregular sparse pattern. This irregularity makes it difficult to fully exploit the sparsity on hardware, as most hardware accelerators (like GPUs and TPUs) are designed to work more efficiently with structured data. Without a regular structure, these accelerators may not be able to skip zero elements as effectively, which can limit the computational savings.

Another challenge is the algorithmic complexity involved in pruning and sparse matrix operations. The process of deciding which weights to prune, particularly in an unstructured manner, requires sophisticated algorithms that must balance model accuracy with computational efficiency. These pruning algorithms can be computationally expensive themselves, and applying them across large models can result in significant overhead. The optimization of sparse matrices also requires specialized techniques that may not always be easy to implement or generalize across different architectures.

Hardware support is another important limitation. Although modern GPUs, TPUs, and FPGAs have specialized features designed to accelerate sparse operations, fully optimizing sparse networks on hardware requires careful alignment between the hardware architecture and the sparsity format. While structured sparsity is easier to leverage on these accelerators, unstructured sparsity remains a challenge, as hardware accelerators may struggle to efficiently handle irregular sparse patterns. Even when hardware is optimized for sparse operations, the overhead associated with sparse matrix storage formats and the need for specialized memory management can still result in suboptimal performance.

Moreover, there is always a trade-off between sparsity and accuracy. Aggressive pruning or low-rank approximation techniques that aggressively reduce the number of parameters can lead to accuracy degradation. Finding the right balance between reducing parameters and maintaining high model performance is a delicate process that requires extensive experimentation. In some cases, introducing too much sparsity can result in a model that is too small or too underfit to achieve high performance.

Additionally, while sparsity can lead to energy savings, energy efficiency is not always guaranteed. Although sparse operations require fewer floating-point operations, the overhead of managing sparse data and ensuring that hardware optimally skips over zero values can introduce additional power consumption. In edge devices or mobile environments with tight power budgets, the benefits of sparsity may be less clear if the overhead associated with sparse data structures and hardware utilization outweighs the energy savings.

Finally, there is a limited applicability of sparsity to certain types of models or tasks. Not all models benefit equally from sparsity, especially those where dense representations are crucial for performance. For example, models in domains such as image segmentation or some types of reinforcement learning may not show significant gains when sparsity is introduced. Additionally, sparsity may not be effective for all hardware platforms, particularly for older or lower-end devices that lack the computational power or specialized features required to take advantage of sparse matrix operations.

10.6.3.6 Sparsity and Other Optimizations

While sparsity in neural networks is a powerful technique for improving computational efficiency and reducing memory usage, its full potential is often realized when it is used alongside other optimization strategies. These optimizations include techniques like pruning, quantization, and efficient model design. Understanding how sparsity interacts with these methods is crucial for effectively combining them to achieve optimal performance (Hoefler, Alistarh, Ben-Nun, Dryden, and Ziogas 2021).

Sparsity and Pruning. Pruning and sparsity are closely related techniques. Pruning is the process of removing unimportant weights or entire components from a network, typically resulting in a sparse model. The goal of pruning is to reduce the number of parameters and operations required during inference, and it inherently leads to sparsity in the model. However, the interaction between pruning and sparsity is not always straightforward.

When pruning is applied, the resulting model may become sparse, but the sparsity pattern, such as whether it is structured or unstructured, affects how effectively the model can be optimized for hardware. For example, structured pruning (e.g., pruning entire filters or layers) typically results in more efficient sparsity, as hardware accelerators like GPUs and TPUs are better equipped to handle regular patterns in sparse matrices (Elsen et al. 2020). Unstructured pruning, on the other hand, can introduce irregular sparsity patterns, which may not be as efficiently processed by hardware, especially when combined with other techniques like quantization.

Pruning methods often rely on the principle of removing weights that have little impact on the model's performance, but when combined with sparsity, they require careful coordination with hardware-specific optimizations. For instance, sparse patterns created by pruning need to align with the underlying hardware architecture to achieve the desired computational savings (Gale, Elsen, and Hooker 2019b).

Sparsity and Quantization. Quantization is another optimization technique that reduces the precision of the model's weights, typically converting them from floating-point numbers to lower-precision integers. When sparsity and quantization are used together, they can complement each other by further reducing the memory footprint and computational cost.

However, the interaction between sparsity and quantization presents unique challenges. While sparsity reduces the number of non-zero elements in a model, quantization reduces the precision of the individual weights. When these two optimizations are applied together, they can lead to significant reductions in both memory usage and computation, but also pose trade-offs in model accuracy (Nagel et al. 2021b). If the sparsity is unstructured, it may exacerbate the challenges of processing the low-precision weights effectively, especially if the hardware does not support irregular sparse matrices efficiently.

Moreover, both sparsity and quantization require hardware that is specifically optimized for these operations. For instance, GPUs and TPUs can accelerate sparse matrix operations, but these gains are amplified when combined with low-precision arithmetic operations. In contrast, CPUs may struggle with the combined overhead of managing sparse and low-precision data simultaneously (Yi Zhang et al. 2021).

Sparsity and Model Design. Efficient model design focuses on creating architectures that are inherently efficient, without the need for extensive post-training optimizations like pruning or quantization. Techniques like depthwise separable convolutions, low-rank approximation, and dynamic computation contribute to sparsity indirectly by reducing the number of parameters or the computational complexity required by a network.

Sparsity enhances the impact of efficient design by reducing the memory and computation requirements even further. For example, using low-rank approximations to compress weight matrices can result in fewer parameters and reduced model size, while sparsity ensures that these smaller models are processed efficiently (Dettmers and Zettlemoyer 2019). Additionally, when applied to models designed with efficient structures, sparsity ensures that the reduction in operations is fully realized during both training and inference.

However, a model designed for efficiency that incorporates sparsity must also be optimized for hardware that supports sparse operations. Without specialized hardware support for sparse data, even the most efficient models can experience suboptimal performance. Therefore, efficient design and sparsity must be aligned with the underlying hardware to ensure that both computational cost and memory usage are minimized (Elsen et al. 2020).

Sparsity and Optimization Challenges. While sparsity can provide significant benefits when combined with pruning, quantization, and efficient model design, there are also challenges in coordinating these techniques. One major challenge is that each optimization method introduces its own set of trade-offs, particularly when it comes to model accuracy. Sparsity can lead to loss of information, while quantization can reduce the precision of the weights, both of which can negatively impact performance if not carefully tuned. Similarly, pruning can result in overly aggressive reductions that degrade accuracy if not managed properly (Labarge, n.d.).

Furthermore, hardware support is a key factor in determining how well these techniques work together. For example, sparsity is more effective when it is structured in a way that aligns with the architecture of the hardware. Hardware accelerators like GPUs and TPUs are optimized for structured sparsity, but may struggle with unstructured patterns or combinations of sparsity and quantization. Achieving optimal performance requires selecting the right combination of sparsity, quantization, pruning, and efficient design, as well as ensuring that the model is aligned with the capabilities of the hardware (Gale, Elsen, and Hooker 2019b).

In summary, sparsity interacts closely with pruning, quantization, and efficient model design. While each of these techniques has its own strengths, combining them requires careful consideration of their impact on model accuracy, computational cost, memory usage, and hardware efficiency. When applied together, these optimizations can lead to significant reductions in both computation and memory usage, but their effectiveness depends on how well they are coordinated and aligned with hardware capabilities. By understanding the synergies and trade-offs between sparsity and other optimization techniques, practitioners can design more efficient models that are well-suited for deployment in real-world, resource-constrained environments.

10.7 AutoML and Model Optimization

As machine learning models grow in complexity, optimizing them for real-world deployment requires balancing multiple factors, including accuracy, efficiency, and hardware constraints. In this chapter, we have explored various optimization techniques, including pruning, quantization, and neural architecture search, each of which targets specific aspects of model efficiency. However, applying these optimizations effectively often requires extensive manual effort, domain expertise, and iterative experimentation.

Automated Machine Learning (AutoML) aims to streamline this process by automating the search for optimal model configurations. AutoML frameworks leverage machine learning algorithms to optimize architectures, hyperparameters, model compression techniques, and other critical parameters, reducing the need for human intervention (F. Hutter, Kotthoff, and Vanschoren 2019b). By systematically exploring the vast design space of possible models, AutoML can improve efficiency while maintaining competitive accuracy, often discovering novel solutions that may be overlooked through manual tuning (Zoph and Le 2017b).

AutoML does not replace the need for human expertise but rather enhances it by providing a systematic and scalable approach to model optimization. As illustrated in Figure 10.30, the key difference between traditional workflows and AutoML is that preprocessing, training and evaluation are automated in the latter. Instead of manually adjusting pruning thresholds, quantization strategies, or architecture designs, practitioners can define high-level objectives, including latency constraints, memory limits, and accuracy targets, and allow AutoML systems to explore configurations that best satisfy these constraints (Feurer et al. 2019).

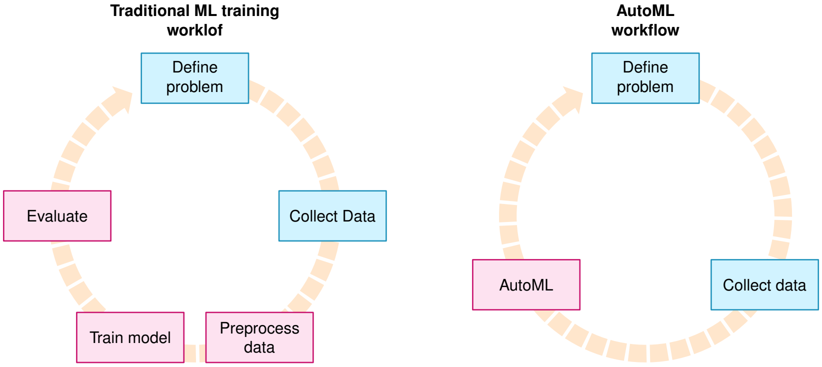


Figure 10.30: Comparing AutoML to the traditional ML training workflow.

We will explore the core aspects of AutoML, starting with the key dimensions of optimization, followed by the methodologies used in AutoML systems, and concluding with challenges and limitations. By the end, we will understand how AutoML serves as an integrative framework that unifies many of the optimization strategies discussed earlier in this chapter.

10.7.1 AutoML Optimizations

AutoML is designed to optimize multiple aspects of a machine learning model, ensuring efficiency, accuracy, and deployability. Unlike traditional approaches that focus on individual techniques, such as quantization for reducing numerical precision or pruning for compressing models, AutoML takes a holistic approach by jointly considering these factors. This enables a more comprehensive search for optimal model configurations, balancing performance with real-world constraints (Yihui He et al. 2018).

One of the primary optimization targets of AutoML is neural network architecture search. Designing an efficient model architecture is a complex process that requires balancing layer configurations, connectivity patterns, and computational costs. NAS automates this by systematically exploring different network structures, evaluating their efficiency, and selecting the most optimal design (Elsken, Metzen, and Hutter 2019b). This process has led to the discovery of architectures such as MobileNetV3 and EfficientNet, which outperform manually designed models on key efficiency metrics (Tan and Le 2019c).

Beyond architecture design, AutoML also focuses on hyperparameter optimization, which plays a crucial role in determining a model's performance. Parameters such as learning rate, batch size, weight decay, and activation functions must be carefully tuned for stability and efficiency. Instead of relying on trial and error, AutoML frameworks employ systematic search strategies, including Bayesian optimization[32], evolutionary algorithms, and adaptive heuristics, to efficiently identify the best hyperparameter settings for a given model and dataset (Bardenet et al. 2015).

Another critical aspect of AutoML is model compression. Techniques such as pruning and quantization help reduce the memory footprint and computational requirements of a model, making it more suitable for deployment on resource-constrained hardware. AutoML frameworks automate the selection of pruning thresholds, sparsity patterns, and quantization levels, optimizing models for both speed and energy efficiency (Jiaxiang Wu et al. 2016). This is particularly important for edge AI applications, where models need to operate with minimal latency and power consumption (Chowdhery et al. 2021).

Finally, AutoML considers deployment-aware optimization, ensuring that the final model is suited for real-world execution. Different hardware platforms impose varying constraints on model execution, such as memory bandwidth limitations, computational throughput, and energy efficiency requirements. AutoML frameworks incorporate hardware-aware optimization techniques, tailoring models to specific devices by adjusting computational workloads, memory access patterns, and execution strategies (H. Cai, Gan, and Han 2020).

Finally, AutoML considers deployment-aware optimization, ensuring that the final model is suited for real-world execution. Different hardware platforms impose varying constraints on model execution, such as memory bandwidth limitations, computational throughput, and energy efficiency requirements. AutoML frameworks incorporate hardware-aware optimization techniques, tailoring models to specific devices by adjusting computational workloads, memory access patterns, and execution strategies.

Optimization across these dimensions enables AutoML to provide a unified framework for enhancing machine learning models, streamlining the process to achieve efficiency without sacrificing accuracy. This holistic approach ensures that models are not only theoretically optimal but also practical for real-world deployment across diverse applications and hardware platforms.

[32] Bayesian Optimization: A strategy for global optimization of black-box functions that is particularly suited for hyperparameter tuning.

10.7.2 Optimization Strategies

AutoML systems optimize machine learning models by systematically exploring different configurations and selecting the most efficient combination of architectures, hyperparameters, and compression strategies. Unlike traditional manual tuning, which requires extensive domain expertise and iterative experimentation, AutoML leverages algorithmic search methods to automate this process. The effectiveness of AutoML depends on how it navigates the vast design space of possible models while balancing accuracy, efficiency, and deployment constraints.

The foundation of AutoML lies in search-based optimization strategies that efficiently explore different configurations. One of the most well-known techniques within AutoML is NAS, which automates the design of machine learning models. NAS frameworks employ methods such as reinforcement learning, evolutionary algorithms, and gradient-based optimization to discover architectures that maximize efficiency while maintaining high accuracy (Zoph and Le 2017b). By systematically evaluating candidate architectures, NAS can identify structures that outperform manually designed models, leading to breakthroughs in efficient machine learning (Real et al. 2019b).

Beyond architecture search, AutoML systems also focus on hyperparameter optimization (HPO), which fine-tunes crucial training parameters such as learning rate, batch size, and weight decay. Instead of relying on grid search or manual tuning, AutoML frameworks employ Bayesian optimization, random search, and adaptive heuristics to efficiently identify the best hyperparameter settings (Feurer et al. 2019). These methods allow AutoML to converge on optimal configurations faster than traditional trial-and-error approaches.

Another key aspect of AutoML is model compression optimization, where pruning and quantization strategies are automatically selected based on deployment requirements. By evaluating trade-offs between model size, latency, and accuracy, AutoML frameworks determine the best way to reduce computational costs while preserving performance. This enables efficient model deployment on resource-constrained devices without extensive manual tuning.

In addition to optimizing model structures and hyperparameters, AutoML also incorporates data processing and augmentation strategies. Training data quality is critical for achieving high model performance, and AutoML frameworks can automatically determine the best preprocessing techniques to enhance generalization. Techniques such as automated feature selection, adaptive augmentation policies, and dataset balancing are employed to improve model robustness without introducing unnecessary computational overhead.

Recent advancements in AutoML have also led to meta-learning approaches[33], where knowledge from previous optimization tasks is leveraged to accelerate the search for new models. By learning from prior experiments, AutoML systems can intelligently navigate the optimization space, reducing the computational cost associated with training and evaluation (Vanschoren 2018). This allows for faster adaptation to new tasks and datasets.

Finally, many modern AutoML frameworks offer end-to-end automation, integrating architecture search, hyperparameter tuning, and model compression into a single pipeline. Platforms such as Google AutoML, Amazon SageMaker

[33] Meta-learning: Learning knowledge from previous tasks to improve future model training efficiency.

Autopilot, and Microsoft Azure AutoML provide fully automated workflows that streamline the entire model optimization process (L. Li et al. 2017).

The integration of these strategies enables AutoML systems to provide a scalable and efficient approach to model optimization, reducing the reliance on manual experimentation. This automation not only accelerates model development but also enables the discovery of novel architectures and configurations that might otherwise be overlooked.

10.7.3 AutoML Challenges and Considerations

While AutoML offers a powerful framework for optimizing machine learning models, it also introduces several challenges and trade-offs that must be carefully considered. Despite its ability to automate model design and hyperparameter tuning, AutoML is not a one-size-fits-all solution. The effectiveness of AutoML depends on computational resources, dataset characteristics, and the specific constraints of a given application.

One of the most significant challenges in AutoML is computational cost. The process of searching for optimal architectures, hyperparameters, and compression strategies requires evaluating numerous candidate models, each of which must be trained and validated. Methods like NAS can be particularly expensive, often requiring thousands of GPU hours to explore a large search space. While techniques such as early stopping, weight sharing, and surrogate models help reduce search costs, the computational overhead remains a major limitation, especially for organizations with limited access to high-performance computing resources.

Another challenge is bias in search strategies, which can influence the final model selection. The optimization process in AutoML is guided by heuristics and predefined objectives, which may lead to biased results depending on how the search space is defined. If the search algorithm prioritizes certain architectures or hyperparameters over others, it may fail to discover alternative configurations that could be more effective for specific tasks. Additionally, biases in training data can propagate through the AutoML process, reinforcing unwanted patterns in the final model.

Generalization and transferability present additional concerns. AutoML-generated models are optimized for specific datasets and deployment conditions, but their performance may degrade when applied to new tasks or environments. Unlike manually designed models, where human intuition can guide the selection of architectures that generalize well, AutoML relies on empirical evaluation within a constrained search space. This limitation raises questions about the robustness of AutoML-optimized models when faced with real-world variability.

Interpretability is another key consideration. Many AutoML-generated architectures and configurations are optimized for efficiency but lack transparency in their design choices. Understanding why a particular AutoML-discovered model performs well can be challenging, making it difficult for practitioners to debug issues or adapt models for specific needs. The black-box nature of some AutoML techniques limits human insight into the underlying optimization process.

Beyond technical challenges, there is also a trade-off between automation and control. While AutoML reduces the need for manual intervention, it also abstracts away many decision-making processes that experts might otherwise fine-tune for specific applications. In some cases, domain knowledge is essential for guiding model optimization, and fully automated systems may not always account for subtle but important constraints imposed by the problem domain.

Despite these challenges, AutoML continues to evolve, with ongoing research focused on reducing computational costs, improving generalization, and enhancing interpretability. As these improvements emerge, AutoML is expected to play an increasingly prominent role in the development of optimized machine learning models, making AI systems more accessible and efficient for a wide range of applications.

10.8 Software and Framework Support

The theoretical understanding of model optimization techniques like pruning, quantization, and efficient numerics is essential, but their practical implementation relies heavily on robust software support. Without extensive framework development and tooling, these optimization methods would remain largely inaccessible to practitioners. For instance, implementing quantization would require manual modification of model definitions and careful insertion of quantization operations throughout the network. Similarly, pruning would involve direct manipulation of weight tensors—tasks that become prohibitively complex as models scale.

The widespread adoption of model optimization techniques has been enabled by significant advances in software frameworks, optimization tools, and hardware integration. Modern machine learning frameworks provide high-level APIs and automated workflows that abstract away much of the complexity involved in applying these optimizations. This software infrastructure makes sophisticated optimization techniques accessible to a broader audience of practitioners, enabling the deployment of efficient models across diverse applications.

Framework support addresses several critical challenges in model optimization:

1. **Implementation Complexity**: Frameworks provide pre-built modules and functions for common optimization techniques, eliminating the need for custom implementations.
2. **Hyperparameter Management**: Tools assist in tuning optimization parameters, such as pruning schedules or quantization bit-widths.
3. **Performance Trade-offs**: Software helps manage the balance between model compression and accuracy through automated evaluation pipelines.
4. **Hardware Compatibility**: Frameworks ensure optimized models remain compatible with target deployment platforms through device-specific code generation and validation.

The support provided by frameworks transforms the theoretical optimization techniques we learned into practical tools that can be readily applied in production environments. This accessibility has been crucial in bridging the gap between academic research and industrial applications, enabling the widespread deployment of efficient machine learning models.

10.8.1 Optimization APIs

Modern machine learning frameworks provide extensive APIs and libraries that enable practitioners to apply optimization techniques without implementing complex algorithms from scratch. These built-in optimizations enhance model efficiency while ensuring adherence to established best practices. Leading frameworks such as TensorFlow, PyTorch, and MXNet offer comprehensive toolkits for model optimization, streamlining the deployment of efficient machine learning systems.

TensorFlow provides robust optimization capabilities through its Model Optimization Toolkit, which facilitates various techniques, including quantization, pruning, and clustering. QAT within the toolkit enables the conversion of floating-point models to lower-precision formats, such as INT8, while preserving model accuracy. The toolkit systematically manages both weight and activation quantization, ensuring consistency across diverse model architectures.

Beyond quantization, TensorFlow's optimization suite includes pruning algorithms that introduce sparsity into neural networks by removing redundant connections at different levels of granularity, from individual weights to entire layers. This flexibility allows practitioners to tailor pruning strategies to their specific requirements. Additionally, weight clustering groups similar weights together to achieve model compression while preserving core functionality. By leveraging these optimization techniques, TensorFlow provides multiple pathways for improving model efficiency beyond traditional quantization.

Similarly, PyTorch offers comprehensive optimization support through built-in modules for quantization and pruning. The `torch.quantization` package provides tools for converting models to lower-precision representations, supporting both post-training quantization and quantization-aware training, as shown in Listing 10.1.

For pruning, PyTorch provides the `torch.nn.utils.prune` module, which supports both unstructured and structured pruning. An example of both pruning strategies is given in Listing 10.2.

These tools integrate seamlessly into PyTorch's training pipelines, enabling efficient experimentation with different optimization strategies.

Built-in optimization APIs offer substantial benefits that make model optimization more accessible and reliable. By providing pre-tested, production-ready tools, these APIs dramatically reduce the implementation complexity that practitioners face when optimizing their models. Rather than having to implement complex optimization algorithms from scratch, developers can leverage standardized interfaces that have been thoroughly vetted.

The consistency provided by these built-in APIs is particularly valuable when working across different model architectures. The standardized interfaces ensure that optimization techniques are applied uniformly, reducing the risk of implementation errors or inconsistencies that could arise from custom solutions.

Listing 10.1: Model preparation for quantization-aware training in PyTorch

```python
import torch
from torch.quantization import QuantStub, DeQuantStub,
     prepare_qat

# Define a model with quantization support
class QuantizedModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.quant = QuantStub()
        self.conv = torch.nn.Conv2d(3, 64, 3)
        self.dequant = DeQuantStub()

    def forward(self, x):
        x = self.quant(x)
        x = self.conv(x)
        return self.dequant(x)

# Prepare model for quantization-aware training
model = QuantizedModel()
model.qconfig = torch.quantization.get_default_qat_qconfig()
model_prepared = prepare_qat(model)
```

Listing 10.2: Weight pruning techniques in PyTorch

```python
import torch.nn.utils.prune as prune

# Apply unstructured pruning
module = torch.nn.Linear(10, 10)
prune.l1_unstructured(module, name='weight', amount=0.3)
# Prune 30% of weights

# Apply structured pruning
prune.ln_structured(module, name='weight', amount=0.5,
                    n=2, dim=0)
```

This standardization helps maintain reliable and reproducible results across different projects and teams.

These frameworks also serve as a bridge between cutting-edge research and practical applications. As new optimization techniques emerge from the research community, framework maintainers incorporate these advances into their APIs, making state-of-the-art methods readily available to practitioners. This continuous integration of research advances ensures that developers have access to the latest optimization strategies without needing to implement them independently.

Furthermore, the comprehensive nature of built-in APIs enables rapid experimentation with different optimization approaches. Developers can easily test various strategies, compare their effectiveness, and iterate quickly to find the optimal configuration for their specific use case. This ability to experiment efficiently is crucial for finding the right balance between model performance and resource constraints.

As model optimization continues to evolve, major frameworks maintain and expand their built-in support, further reducing barriers to efficient model deployment. The standardization of these APIs has played a crucial role in democratizing access to model efficiency techniques while ensuring high-quality implementations remain consistent and reliable.

10.8.2 Hardware Optimization Libraries

Hardware optimization libraries in modern machine learning frameworks enable efficient deployment of optimized models across different hardware platforms. These libraries integrate directly with training and deployment pipelines to provide hardware-specific acceleration for various optimization techniques across model representation, numerical precision, and architectural efficiency dimensions.

For model representation optimizations like pruning, libraries such as TensorRT, XLA, and OpenVINO provide sparsity-aware acceleration through optimized kernels that efficiently handle sparse computations. TensorRT specifically supports structured sparsity patterns, allowing models trained with techniques like two-out-of-four structured pruning to run efficiently on NVIDIA GPUs. Similarly, TPUs leverage XLA's sparse matrix optimizations, while FPGAs enable custom sparse execution through frameworks like Vitis AI.

Knowledge distillation benefits from hardware-aware optimizations that help compact student models achieve high inference efficiency. Libraries like TensorRT, OpenVINO, and SNPE optimize distilled models for low-power execution, often combining distillation with quantization or architectural restructuring to meet hardware constraints. For models discovered through neural architecture search (NAS), frameworks such as TVM and TIMM provide compiler support to tune the architectures for various hardware backends.

In terms of numerical precision optimization, these libraries offer extensive support for both PTQ and QAT. TensorRT and TensorFlow Lite implement INT8 and INT4 quantization during model conversion, reducing computational complexity while leveraging specialized hardware acceleration on mobile SoCs and edge AI chips. NVIDIA TensorRT incorporates calibration-based quantization using representative datasets to optimize weight and activation scaling.

More granular quantization approaches like channelwise and groupwise quantization are supported in frameworks such as SNPE and OpenVINO. Dynamic quantization capabilities in PyTorch and ONNX Runtime enable runtime activation quantization, making models adaptable to varying hardware conditions. For extreme precision reduction, techniques like binarization and ternarization are optimized through libraries such as CMSIS-NN, enabling efficient execution of binary-weight models on ARM Cortex-M microcontrollers.

Architectural efficiency techniques integrate tightly with hardware-specific execution frameworks. TensorFlow XLA and TVM provide operator-level tuning through aggressive fusion and kernel reordering, improving efficiency across GPUs, TPUs, and edge devices. Dynamic computation approaches like early exit architectures and conditional computation are supported by custom execution runtimes that optimize control flow.

The widespread support for sparsity-aware execution spans multiple hardware platforms. NVIDIA GPUs utilize specialized sparse tensor cores for accelerating structured sparse models, while TPUs implement hardware-level sparse matrix optimizations. On FPGAs, vendor-specific compilers like Vitis AI enable custom sparse computations to be highly optimized.

This comprehensive integration of hardware optimization libraries with machine learning frameworks enables developers to effectively implement pruning, quantization, NAS, dynamic computation, and sparsity-aware execution while ensuring optimal adaptation to target hardware. The ability to optimize across multiple dimensions, including model representation, numerical precision, and architectural efficiency, is crucial for deploying machine learning models efficiently across diverse platforms.

10.8.3 Optimization Visualization

Model optimization techniques fundamentally alter model structure and numerical representations, but their impact can be difficult to interpret without visualization tools. Dedicated visualization frameworks and libraries help practitioners gain insights into how pruning, quantization, and other optimizations affect model behavior. These tools provide graphical representations of sparsity patterns, quantization error distributions, and activation changes, making optimization more transparent and controllable.

10.8.3.1 Quantization Visualization

Quantization reduces numerical precision, introducing rounding errors that can impact model accuracy. Visualization tools provide direct insight into how these errors are distributed, helping diagnose and mitigate precision-related performance degradation.

One commonly used technique is quantization error histograms, which depict the distribution of errors across weights and activations. These histograms reveal whether quantization errors follow a Gaussian distribution or contain outliers, which could indicate problematic layers. TensorFlow's Quantization Debugger and PyTorch's FX Graph Mode Quantization tools allow users to analyze such histograms and compare error patterns between different quantization methods.

Activation visualizations also help detect overflow issues caused by reduced numerical precision. Tools such as ONNX Runtime's quantization visualization utilities and NVIDIA's TensorRT Inspector allow practitioners to color-map activations before and after quantization, making saturation and truncation issues visible. This enables calibration adjustments to prevent excessive information loss, preserving numerical stability. For example, Figure 10.31 is a color mapping of the AlexNet convolutional kernels.

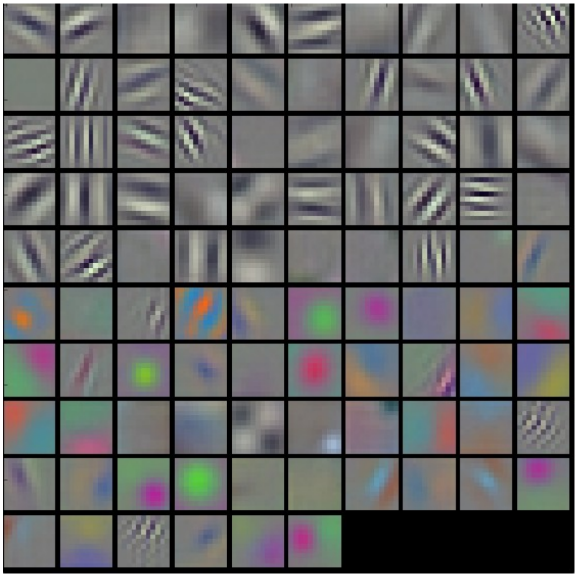


Figure 10.31: Color mapping of activations. Source: Krizhevsky, Sutskever, and Hinton (2017c).

Beyond static visualizations, tracking quantization error over the training process is essential. Monitoring mean squared quantization error (MSQE) during quantization-aware training (QAT) helps identify divergence points where numerical precision significantly impacts learning. TensorBoard and PyTorch's quantization debugging APIs provide real-time tracking, highlighting instability during training.

By integrating these visualization tools into optimization workflows, practitioners can identify and correct issues early, ensuring optimized models maintain both accuracy and efficiency. These empirical insights provide a deeper understanding of how sparsity, quantization, and architectural optimizations affect models, guiding effective model compression and deployment strategies.

10.8.3.2 Sparsity Visualization

Sparsity visualization tools provide detailed insight into pruned models by mapping out which weights have been removed and how sparsity is distributed across different layers. Frameworks such as TensorBoard (for TensorFlow) and Netron (for ONNX) allow users to inspect pruned networks at both the layer and weight levels.

One common visualization technique is sparsity heat maps, where color gradients indicate the proportion of weights removed from each layer. Layers with higher sparsity appear darker, revealing the model regions most impacted by pruning, as shown in Figure 10.32. This type of visualization transforms pruning from a black-box operation into an interpretable process, enabling practitioners to better understand and control sparsity-aware optimizations.

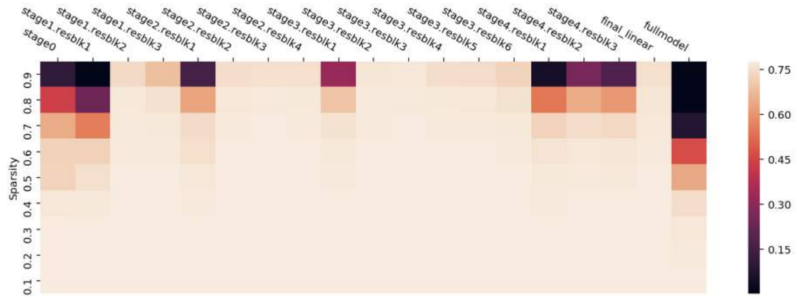


Figure 10.32: Sparse network heat map. Source: Numenta.

Beyond static snapshots, trend plots track sparsity progression across multiple pruning iterations. These visualizations illustrate how global model sparsity evolves, often showing an initial rapid increase followed by more gradual refinements. Tools like TensorFlow's Model Optimization Toolkit and SparseML's monitoring utilities provide such tracking capabilities, displaying per-layer pruning levels over time. These insights allow practitioners to fine-tune pruning strategies by adjusting sparsity constraints for individual layers.

Libraries such as DeepSparse's visualization suite and PyTorch's pruning utilities enable the generation of these visualization tools, helping analyze how pruning decisions affect different model components. By making sparsity data visually accessible, these tools help practitioners optimize their models more effectively.

10.9 Conclusion

This chapter has explored the multifaceted landscape of model optimization, a critical process for translating machine learning advancements into practical, real-world systems. We began by recognizing the inherent tension between model accuracy and efficiency, driven by constraints such as computational cost, memory limitations, and energy consumption. This necessitates a systematic approach to refining models, ensuring they remain effective while operating within the boundaries of real-world deployment environments.

We examined three core dimensions of model optimization: optimizing model representation, numerical precision, and architectural efficiency. Within each dimension, we delved into specific techniques, such as pruning, knowledge distillation, quantization, and dynamic computation, highlighting their trade-offs and practical considerations. We also emphasized the importance of hardware-aware model design, recognizing that aligning model architectures with the underlying hardware capabilities is crucial for maximizing performance and efficiency.

Finally, we explored AutoML as a holistic approach to model optimization, automating many of the tasks that traditionally require manual effort and expertise. AutoML frameworks offer a unified approach to architecture search, hyperparameter tuning, model compression, and data processing, streamlining the optimization process and potentially leading to novel solutions that might be overlooked through manual exploration.

As machine learning continues to evolve, model optimization will remain a critical area of focus. The ongoing development of new techniques, coupled with advancements in hardware and software infrastructure, will further enhance our ability to deploy efficient, scalable, and robust AI systems. By understanding the principles and practices of model optimization, practitioners can effectively bridge the gap between theoretical advancements and practical applications, unlocking the full potential of machine learning to address real-world challenges.

10.10 Resources

Slides

- *Coming soon.*

Videos

- *Coming soon.*

Exercises

- *Coming soon.*

Chapter 11

AI Acceleration

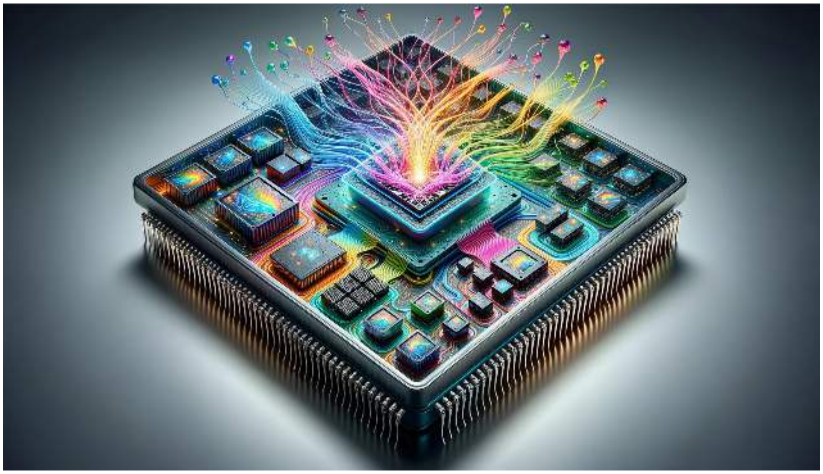


Figure 11.1: *DALL·E 3 Prompt: Create an intricate and colorful representation of a System on Chip (SoC) design in a rectangular format. Showcase a variety of specialized machine learning accelerators and chiplets, all integrated into the processor. Provide a detailed view inside the chip, highlighting the rapid movement of electrons. Each accelerator and chiplet should be designed to interact with neural network neurons, layers, and activations, emphasizing their processing speed. Depict the neural networks as a network of interconnected nodes, with vibrant data streams flowing between the accelerator pieces, showcasing the enhanced computation speed.*

Purpose

How does hardware acceleration impact machine learning system performance, and what principles should ML engineers understand to effectively design and deploy systems?

Machine learning systems has driven a fundamental shift in computer architecture. Traditional processors, designed for general-purpose computing, prove inefficient for the repeated mathematical operations and data movement patterns in neural networks. Modern accelerators address this challenge by matching hardware structures to ML computation patterns. These accelerators introduce fundamental trade-offs in performance, power consumption, and flexibility. Effective utilization of hardware acceleration requires an understanding of these trade-offs, as well as the architectural principles that govern accelerator design. By optimizing and learning to map models effectively for specific hardware platforms, engineers can balance computational efficiency.

Learning Objectives

- Understand the historical context of hardware acceleration.
- Identify key AI compute primitives and their role in model execution.
- Explain the memory hierarchy and its impact on AI accelerator performance.
- Describe strategies for mapping neural networks to hardware.
- Analyze the role of compilers and runtimes in optimizing AI workloads.
- Compare single-chip and multi-chip AI architectures.

11.1 Overview

Machine learning has driven a fundamental shift in computer architecture, pushing beyond traditional general-purpose processors toward specialized acceleration. The computational demands of modern machine learning models exceed the capabilities of conventional CPUs, which were designed for sequential execution. Instead, machine learning workloads exhibit massive parallelism, high memory bandwidth requirements, and structured computation patterns that demand purpose-built hardware for efficiency and scalability. Machine Learning Accelerators (ML Accelerators) have emerged as a response to these challenges.

Definition of ML Accelerator

Machine Learning Accelerator (ML Accelerator) refers to a *specialized computing hardware* designed to *efficiently execute machine learning workloads*. These accelerators optimize *matrix multiplications, tensor operations, and data movement*, enabling *high-throughput and energy-efficient* computation. ML accelerators operate at various *power and performance scales*, ranging from *edge devices with milliwatt-level consumption* to *data center-scale accelerators requiring kilowatts of power*. They are specifically designed to address *the computational and memory demands* of machine learning models, often incorporating *optimized memory hierarchies, parallel processing units, and custom instruction sets* to maximize performance. ML accelerators are widely used in *training, inference, and real-time AI applications* across cloud, edge, and embedded systems.

Unlike CPUs and GPUs,[0] which were originally designed for general-purpose computing and graphics, ML accelerators are optimized for tensor operations, matrix multiplications, and memory-efficient execution—the core computations that drive deep learning. These accelerators span a wide range of power and performance envelopes, from energy-efficient edge devices to large-scale

[0] While GPUs were initially designed for digital image processing and accelerating computer graphics, they proved useful for non-graphic calculations. The parallel structure of GPUs is useful in processing the color of thousands of pixels simultaneously. Many computational problems in ML such as matrix multiplication or processing of large datasets share similar characteristics with graphic processing.

data center accelerators. Their architectures integrate custom processing elements, optimized memory hierarchies, and domain-specific execution models, enabling high-performance training and inference.

As ML models have grown in size and complexity, hardware acceleration has evolved to keep pace. The shift from von Neumann architectures[1] to specialized accelerators reflects a broader trend in computing: reducing the cost of data movement, increasing parallelism, and tailoring hardware to domain-specific workloads. Moving data across memory hierarchies often consumes more energy than computation itself, making efficient memory organization and computation placement critical to overall system performance.

This chapter explores AI acceleration from a systems perspective, examining how computational models, hardware optimizations, and software frameworks interact to enable efficient execution. It covers key operations like matrix multiplications and activation functions, the role of memory hierarchies in data movement, and techniques for mapping neural networks to hardware. The discussion extends to compilers, scheduling strategies, and runtime optimizations, highlighting their impact on performance. Finally, it addresses the challenges of scaling AI systems from single-chip accelerators to multi-chip and distributed architectures, integrating real-world examples to illustrate effective AI acceleration.

11.2 Hardware Evolution

The progression of computing architectures follows a recurring pattern: as computational workloads grow in complexity, general-purpose processors become increasingly inefficient, prompting the development of specialized hardware accelerators. This transition is driven by the need for higher computational efficiency, reduced energy consumption, and optimized execution of domain-specific workloads. Machine learning acceleration is the latest stage in this ongoing evolution, following a well-established trajectory observed in prior domains such as floating-point arithmetic, graphics processing, and digital signal processing.

This evolution is not just of academic interest—it provides essential context for understanding how modern ML accelerators like GPUs with tensor cores, Google's TPUs, and Apple's Neural Engine came to be. These technologies now power widely deployed applications such as real-time language translation, image recognition, and personalized recommendations. The architectural strategies enabling such capabilities are deeply rooted in decades of hardware specialization.

At the heart of this transition is hardware specialization, which enhances performance and efficiency by optimizing frequently executed computational patterns through dedicated circuit implementations. While this approach leads to significant gains, it also introduces trade-offs in flexibility, silicon area utilization, and programming complexity. As computing demands continue to evolve, specialized accelerators must balance these factors to deliver sustained improvements in efficiency and performance.

Building on this historical trajectory, the evolution of hardware specialization provides a perspective for understanding modern machine learning acceler-

[1] von Neumann Architecture: A computing model where programs and data share the same memory, leading to a bottleneck in data transfer between the processor and memory, known as the von Neumann bottleneck.

ators. Many of the principles that shaped the development of early floating-point and graphics accelerators now inform the design of AI-specific hardware. Examining these past trends offers a systematic framework for analyzing contemporary approaches to AI acceleration and anticipating future developments in specialized computing.

11.2.1 Specialized Computing

The transition toward specialized computing architectures arises from the fundamental limitations of general-purpose processors. Early computing systems relied on central processing units (CPUs) to execute all computational tasks sequentially, following a one-size-fits-all approach. However, as computing workloads diversified and grew in complexity, certain operations, especially floating-point arithmetic, emerged as critical performance bottlenecks that could not be efficiently handled by CPUs alone. These fundamental inefficiencies prompted the development of specialized hardware architectures designed to accelerate specific computational patterns (Flynn 1966).

One of the earliest examples of hardware specialization was the Intel 8087 mathematics coprocessor, introduced in 1980. This floating-point unit (FPU) was designed to offload arithmetic-intensive computations from the main CPU, dramatically improving performance for scientific and engineering applications. The 8087 demonstrated unprecedented efficiency, achieving performance gains of up to 100× for floating-point operations compared to software-based implementations on general-purpose processors (Fisher 1981). This milestone established a fundamental principle in computer architecture: carefully designed hardware specialization could provide order-of-magnitude improvements for well-defined, computationally intensive tasks.

The success of floating-point coprocessors led to their eventual integration into mainstream processors. For example, the Intel 486DX, released in 1989, incorporated an on-chip floating-point unit, eliminating the need for an external coprocessor. This integration not only improved processing efficiency but also marked a recurring pattern in computer architecture: successful specialized functions tend to become standard features in future generations of general-purpose processors (D. A. Patterson and Hennessy 2021c).

The principles established through early floating-point acceleration continue to influence modern hardware specialization. These include:

1. Identification of computational bottlenecks through workload analysis
2. Development of specialized circuits for frequent operations
3. Creation of efficient hardware-software interfaces
4. Progressive integration of proven specialized functions

This progression from domain-specific specialization to general-purpose integration has played a central role in shaping modern computing architectures. As computational workloads expanded beyond arithmetic operations, these same fundamental principles were applied to new domains, such as graphics processing, digital signal processing, and ultimately, machine learning acceleration. Each of these domains introduced specialized architectures tailored to their unique computational requirements, establishing hardware specialization as a cornerstone strategy for advancing computing performance and efficiency in increasingly complex workloads.

The evolution of specialized computing hardware follows a consistent trajectory, wherein architectural innovations are introduced to mitigate emerging computational bottlenecks and are eventually incorporated into mainstream computing platforms. As illustrated in Figure 11.2, each computing era gave rise to accelerators that addressed the dominant workload characteristics of the time. These developments have not only advanced architectural efficiency but have also shaped the foundation upon which contemporary machine learning systems are built. The computational capabilities required for tasks such as real-time language translation, personalized recommendations, and on-device inference rely on the foundational principles and architectural innovations established in earlier domains, including floating-point computation, graphics processing, and digital signal processing.

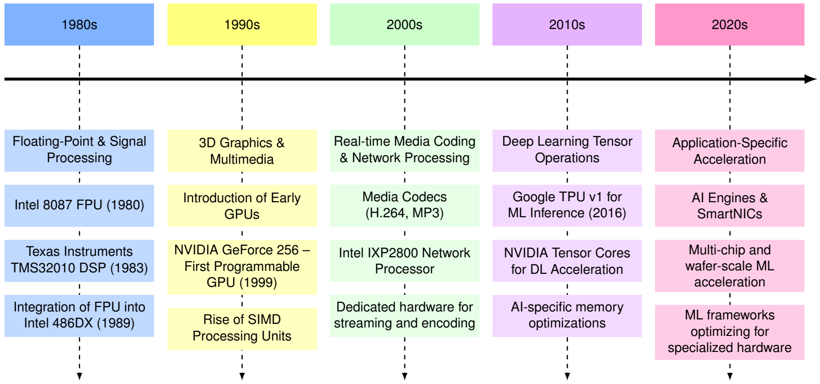

1980s
Floating-Point & Signal Processing
Intel 8087 FPU (1980)
Texas Instruments TMS32010 DSP (1983)
Integration of FPU into Intel 486DX (1989)
1990s
3D Graphics & Multimedia
Introduction of Early GPUs
NVIDIA GeForce 256 – First Programmable GPU (1999)
Rise of SIMD Processing Units
2000s
Real-time Media Coding & Network Processing
Media Codecs (H.264, MP3)
Intel IXP2800 Network Processor
Dedicated hardware for streaming and encoding
2010s
Deep Learning Tensor Operations
Google TPU v1 for ML Inference (2016)
NVIDIA Tensor Cores for DL Acceleration
AI-specific memory optimizations
2020s
Application-Specific Acceleration
AI Engines & SmartNICs
Multi-chip and wafer-scale ML acceleration
ML frameworks optimizing for specialized hardware

Figure 11.2: Evolution of specialized computing hardware.

11.2.2 Specialized Computing Expansion

The principles established through floating-point acceleration provided a blueprint for addressing emerging computational challenges. As computing applications diversified, new computational patterns emerged that exceeded the capabilities of general-purpose processors. This expansion of specialized computing manifested across multiple domains, each contributing unique insights to hardware acceleration strategies.

Graphics processing emerged as a significant driver of hardware specialization in the 1990s. Early graphics accelerators focused on specific operations like bitmap transfers and polygon filling. The introduction of programmable graphics pipelines with NVIDIA's GeForce 256 in 1999 represented a crucial advancement in specialized computing. Graphics Processing Units (GPUs) demonstrated how parallel processing architectures could efficiently handle data-parallel workloads. For example, in 3D rendering tasks like texture mapping and vertex transformation, GPUs achieved 50-100× speedups over CPU implementations. By 2004, GPUs could process over 100 million polygons per second—tasks that would overwhelm even the fastest CPUs of the time (Owens et al. 2008).

Digital Signal Processing (DSP) represents another fundamental domain of hardware specialization. DSP processors introduced architectural innovations specifically designed for efficient signal processing operations. These included specialized multiply-accumulate units, circular buffers, and parallel data paths optimized for filtering and transform operations. Texas Instruments' TMS32010, introduced in 1983, established how domain-specific instruction sets and memory architectures could dramatically improve performance for signal processing applications (Lyons 2011).

Network processing introduced additional patterns of specialization. Network processors developed unique architectures to handle packet processing at line rate, incorporating multiple processing cores, specialized packet manipulation units, and sophisticated memory management systems. Intel's IXP2800 network processor demonstrated how multiple levels of hardware specialization could be combined to address complex processing requirements.

These diverse domains of specialization shared several common themes:

1. Identification of domain-specific computational patterns
2. Development of specialized processing elements and memory hierarchies
3. Creation of domain-specific programming models
4. Progressive evolution toward more flexible architectures

This period of expanding specialization demonstrated that hardware acceleration strategies could successfully address diverse computational requirements. The GPU's success in parallelizing 3D graphics pipelines directly enabled its later adoption for training deep neural networks, such as AlexNet in 2012, which famously ran on consumer-grade NVIDIA GPUs. DSP innovations in low-power signal processing helped pave the way for real-time inference on edge devices, such as voice assistants and wearables. These domains not only informed ML hardware designs but also proved that accelerators could be deployed across both cloud and embedded contexts—a lesson that continues to shape today's AI ecosystem.

11.2.3 Domain-Specific Architectures

The emergence of domain-specific architectures (DSA) marks a fundamental shift in computer system design, driven by two key factors: the breakdown of traditional scaling laws and the increasing computational demands of specialized workloads. The slowdown of Moore's Law, which previously ensured predictable enhancements in transistor density every 18 to 24 months, and the end of Dennard scaling, which permitted frequency increases without corresponding power increases, created a critical performance and efficiency bottleneck in general-purpose computing. As John Hennessy and David Patterson noted in their 2017 Turing Lecture (John L. Hennessy and Patterson 2019), these limitations signaled the onset of a new era in computer architecture—one centered on domain-specific solutions that optimize hardware for specialized workloads.

Historically, improvements in processor performance relied on semiconductor process scaling and increasing clock speeds. However, as power density limitations restricted further frequency scaling, and as transistor miniaturization faced increasing physical and economic constraints, architects were forced to explore alternative approaches to sustain computational growth. The result was a shift toward domain-specific architectures, which dedicate silicon resources to optimize computation for specific application domains, trading flexibility for efficiency. Domain-specific architectures achieve superior performance and energy efficiency through several key principles:

1. **Customized datapaths**: Design processing paths specifically optimized for target application patterns, enabling direct hardware execution of common operations. For example, matrix multiplication units in AI accelerators implement systolic arrays tailored for neural network computations.
2. **Specialized memory hierarchies**: Optimize memory systems around domain-specific access patterns and data reuse characteristics. This includes custom cache configurations, prefetching logic, and memory controllers tuned for expected workloads.
3. **Reduced instruction overhead**: Implement domain-specific instruction sets that minimize decode and dispatch complexity by encoding common operation sequences into single instructions. This improves both performance and energy efficiency.
4. **Direct hardware implementation**: Create dedicated circuit blocks that natively execute frequently used operations without software intervention. This eliminates instruction processing overhead and maximizes throughput.

Perhaps the best-known example of success in domain-specific architectures is modern smartphones. Introduced in the late 2000s, modern smartphones can decode 4K video at 60 frames per second while consuming just a few watts of power—even though video processing requires billions of operations per second. This remarkable efficiency is achieved through dedicated hardware video codecs that implement industry standards such as H.264/AVC (introduced in 2003) and H.265/HEVC (finalized in 2013) (Sullivan et al. 2012). These specialized circuits offer 100–1000× improvements in both performance and power efficiency compared to software-based decoding on general-purpose processors.

The trend toward specialization continues to accelerate, with new architectures emerging for an expanding range of domains. Genomics processing, for example, benefits from custom accelerators that optimize sequence alignment and variant calling, reducing the time required for DNA analysis (Shang, Wang, and Liu 2018). Similarly, blockchain computation has given rise to application-specific integrated circuits (ASICs) optimized for cryptographic hashing, dramatically increasing the efficiency of mining operations (Bedford Taylor 2017). These examples illustrate that domain-specific architecture is not merely a transient trend but a fundamental transformation in computing systems, offering tailored solutions that address the growing complexity and diversity of modern computational workloads.

11.2.4 ML in Computational Domains

Machine learning has emerged as one of the most computationally demanding fields, demonstrating the need for dedicated hardware that targets its unique characteristics. Domain-specific architectures, previously developed for video codecs or other specialized tasks, have now expanded to meet the challenges posed by ML workloads. These specialized designs optimize the execution of dense matrix operations and manage data movement efficiently, a necessity given the inherent memory bandwidth[2] limitations.

A key distinction in ML is the differing requirements between training and inference. Training demands both forward and backward propagation, with high numerical precision (e.g., FP32 or FP16) to ensure stable gradient updates and convergence, while inference can often operate at lower precision (e.g., INT8) without major accuracy loss. This variance not only drives the need for mixed-precision arithmetic hardware but also allows optimizations that improve throughput and energy efficiency—often achieving 4–8× gains.

The computational foundation of modern ML accelerators is built on common patterns such as dense matrix multiplications and consistent data-flow patterns. These operations underpin architectures like GPUs with tensor cores and Google's Tensor Processing Unit (TPU). While GPUs extended their original graphics capabilities to handle ML tasks via parallel execution and specialized memory hierarchies, TPUs take a more focused approach. For instance, the TPU's systolic array architecture is tailored to excel at matrix multiplication, effectively aligning hardware performance with the mathematical structure of neural networks.

11.2.5 Application-Specific Accelerators

The shift toward application-specific hardware is evident in how these accelerators are designed for both high-powered data centers and low-power edge devices. In data centers, powerful training accelerators can reduce model development times from weeks to days, thanks to their finely-tuned compute engines and memory systems. Conversely, edge devices benefit from inference engines that deliver millisecond-level responses while consuming very little power.

The success of these dedicated solutions reinforces a broader trend—hardware specialization adapts to the computational demands of evolving applications. By focusing on the core operations of machine learning, from matrix multiplications to flexible numerical precision, application-specific accelerators ensure that systems remain efficient, scalable, and ready to meet future advancements.

The evolution of specialized hardware architectures illustrates a fundamental principle in computing systems: as computational patterns emerge and mature, hardware specialization follows to achieve optimal performance and energy efficiency. This progression is particularly evident in machine learning acceleration, where domain-specific architectures have evolved to meet the increasing computational demands of machine learning models. Unlike general-purpose processors, which prioritize flexibility, specialized accelerators optimize execution for well-defined workloads, balancing performance, energy efficiency, and integration with software frameworks.

[2] Memory Bandwidth: The rate at which data can be read from or written to memory by a processor, influencing performance in data-intensive operations.

Table 11.1 summarizes key milestones in the evolution of hardware specialization, emphasizing how each era produced architectures tailored to the prevailing computational demands. While these accelerators initially emerged to optimize domain-specific workloads, including floating-point operations, graphics rendering, and media processing, they also introduced architectural strategies that persist in contemporary systems. Notably, the specialization principles outlined in earlier generations now underpin the design of modern AI accelerators. Understanding this historical trajectory provides essential context for analyzing how hardware specialization continues to enable scalable, efficient execution of machine learning workloads across diverse deployment environments.

Table 11.1: Evolution of hardware specialization across computing eras.

Era	Computational Pattern	Architecture Examples	Key Characteristics
1980s	Floating-Point & Signal Processing	FPU, DSP	Single-purpose engines Focused instruction sets Coprocessor interfaces
1990s	3D Graphics & Multimedia	GPU, SIMD Units	Many identical compute units Regular data patterns Wide memory interfaces
2000s	Real-time Media Coding	Media Codecs, Network Processors	Fixed-function pipelines High throughput processing Power-performance optimization
2010s	Deep Learning Tensor Operations	TPU, GPU Tensor Cores	Matrix multiplication units Massive parallelism Memory bandwidth optimization
2020s	Application-Specific Acceleration	ML Engines, Smart NICs, Domain Accelerators	Workload-specific datapaths Customized memory hierarchies Application-optimized designs

This historical progression reveals a recurring pattern: each wave of hardware specialization responded to a computational bottleneck—be it graphics rendering, media encoding, or neural network inference. What distinguishes the 2020s is not just specialization, but its pervasiveness: AI accelerators now underpin everything from product recommendations on YouTube to object detection in autonomous vehicles. Unlike earlier accelerators, today's AI hardware must integrate tightly with dynamic software frameworks and scale across cloud-to-edge deployments. The table illustrates not just the past but also the trajectory toward increasingly tailored, high-impact computing platforms.

In the case of AI acceleration, this transition has introduced challenges that extend well beyond the confines of hardware design. Machine learning accelerators must integrate seamlessly into comprehensive ML workflows by aligning with optimizations at multiple levels of the computing stack. To achieve this, they are required to operate effectively with widely adopted frameworks such as TensorFlow, PyTorch, and JAX, thereby ensuring that deployment is smooth and consistent across varied hardware platforms. In tandem with this, compiler and runtime support become essential; advanced optimization techniques, such as graph-level transformations, kernel fusion, and memory scheduling, are critical for harnessing the full potential of these specialized accelerators.

Moreover, scalability presents an ongoing demand as AI accelerators are deployed in diverse environments ranging from high-throughput data centers to resource-constrained edge and mobile devices, necessitating tailored performance tuning and energy efficiency strategies. Finally, the integration of such accelerators into heterogeneous computing environments underscores the importance of interoperability, ensuring that these specialized units can function in concert with conventional CPUs and GPUs in distributed systems.

The emergence of AI accelerators is therefore not simply a matter of hardware optimization but also a system-level transformation, where improvements in computation must be tightly coupled with advances in compilers, software frameworks, and distributed computing strategies. Understanding these principles is essential for designing and deploying efficient machine learning systems. The following sections explore how modern ML accelerators address these challenges, focusing on their architectural approaches, system-level optimizations, and integration into the broader machine learning ecosystem.

11.3 AI Compute Primitives

Modern neural networks are built upon a small number of core computational patterns. Regardless of the layer type, whether fully connected, convolutional, or attention-based, the underlying operation typically involves multiplying input values by learned weights and accumulating the results. This repeated multiply-accumulate process dominates neural network execution and defines the arithmetic foundation of AI workloads. The regularity and frequency of these operations have led to the development of AI compute primitive*: hardware-level abstractions optimized to execute these core computations with high efficiency.

Unlike traditional software applications, which often involve irregular control flow and diverse instruction types, neural networks exhibit highly structured, data-parallel computations applied across large arrays. This characteristic enables architectural simplifications and optimizations, where hardware is tailored to the consistent patterns in AI execution. These patterns emphasize parallelism, predictable data reuse, and fixed operation sequences—making them ideal candidates for specialized accelerator design. AI compute primitives distill these patterns into reusable architectural units that support high-throughput and energy-efficient execution.

This decomposition is illustrated in Listing 11.1, which defines a dense layer at the framework level.

Listing 11.1: Declarative creation of dense layer

```
dense = Dense(512)(input_tensor)
```

This high-level call expands into mathematical operations is shown in Listing 11.2.

At the processor level, the computation reduces to nested loops that multiply inputs and weights, sum the results, and apply a nonlinear function, as shown in Listing 11.3.

Listing 11.2: Breaking down layer computation into primitives

```
output = matmul(input_weights) + bias
output = activation(output)
```

Listing 11.3: Processor-level nested loop computation

```
for n in range(batch_size):
    for m in range(output_size):
        sum = bias[m]
        for k in range(input_size):
            sum += input[n,k] * weights[k,m]
        output[n,m] = activation(sum)
```

This transformation, from framework-level abstraction to processor-level implementation, reveals four essential computational characteristics. First, data-level parallelism enables simultaneous execution across inputs. Second, structured matrix operations define the computational workload and guide the need for dedicated datapaths. Third, predictable data movement patterns drive memory system design to minimize latency and maximize reuse. Fourth, frequent nonlinear transformations motivate hardware support for activation and normalization functions.

The design of AI compute primitives is guided by three architectural criteria. First, the primitive must be used frequently enough to justify dedicated hardware resources. Second, its specialized implementation must offer substantial performance or energy efficiency gains relative to general-purpose alternatives. Third, the primitive must remain stable across generations of neural network architectures to ensure long-term applicability. These considerations shape the inclusion of primitives such as vector operations, matrix operations, and special function units in modern ML accelerators. Together, they serve as the architectural foundation for efficient and scalable neural network execution.

11.3.1 Vector Operations

Vector operations provide the first level of hardware acceleration by processing multiple data elements simultaneously. This parallelism exists at multiple scales, from individual neurons to entire layers, making vector processing essential for efficient neural network execution. By examining how framework-level code translates to hardware instructions, we can understand the critical role of vector processing in neural accelerators.

11.3.1.1 Framework-Hardware Execution

Machine learning frameworks hide hardware complexity through high-level abstractions. These abstractions decompose into progressively lower-level operations, revealing opportunities for hardware acceleration. One such abstraction is shown in Listing 11.4, which illustrates the execution flow of a linear layer.

Listing 11.4: Framework Level: What ML developers write

```
layer = nn.Linear(256, 512)  # Layer transforms 256 inputs to
                             # 512 outputs
output = layer(input_tensor) # Process a batch of inputs
```

This abstraction represents a fully connected layer that transforms input features through learned weights. As shown in Listing 11.5 the framework translates this high-level expression into mathematical operations.

Listing 11.5: Internal mathematical representation of a linear layer

```
Z = matmul(weights, input) + bias # Each output needs all inputs
output = activation(Z)            # Transform each result
```

These mathematical operations decompose into explicit computational steps during processor execution. See Listing 11.6 for an illustration of these multiply-accumulate operations.

Listing 11.6: Loop-based execution of a linear layer

```
for batch in range(32):             # Process 32 samples at once
    for out_neuron in range(512):   # Compute each output neuron
        sum = 0.0
        for in_feature in range(256): # Each output needs
                                      # all inputs
            sum += input[batch, in_feature] *
                         weights[out_neuron, in_feature]
        output[batch, out_neuron] = activation(sum +
                                    bias[out_neuron])
```

11.3.1.2 Sequential Scalar Execution

Traditional scalar processors[3] execute these operations sequentially, processing individual values one at a time. For the linear layer example above with a batch of 32 samples, computing the outputs requires over 4 million multiply-accumulate operations. Each operation involves loading an input value and a weight value, multiplying them, and accumulating the result. This sequential approach becomes highly inefficient when processing the massive number of identical operations required by neural networks.

[3] Scalar Processor: A scalar processor handles one data element per cycle, executing operations sequentially rather than in parallel.

11.3.1.3 Parallel Vector Execution

Vector processing units transform this execution pattern by operating on multiple data elements simultaneously. As shown in Listing 11.7, the RISC-V assembly code demonstrates modern vector processing.

Listing 11.7: RISC-V vectorized multiply-accumulate loop

```
vsetvli t0, a0, e32   # Process 8 elements at once
loop_batch:
    loop_neuron:
        vxor.vv v0, v0, v0    # Clear 8 accumulators
        loop_feature:
            vle32.v v1, (in_ptr)    # Load 8 inputs together
            vle32.v v2, (wt_ptr)    # Load 8 weights together
            vfmacc.vv v0, v1, v2    # 8 multiply-adds at once
            add in_ptr, in_ptr, 32  # Move to next 8 inputs
            add wt_ptr, wt_ptr, 32  # Move to next 8 weights
            bnez feature_cnt, loop_feature
```

This vector implementation processes eight data elements in parallel, reducing both computation time and energy consumption. Vector load instructions transfer eight values simultaneously, maximizing memory bandwidth utilization. The vector multiply-accumulate instruction processes eight pairs of values in parallel, dramatically reducing the total instruction count from over 4 million to approximately 500,000.

To clarify how vector instructions map to common deep learning patterns, Table 11.2 introduces key vector operations and their typical applications in neural network computation. These operations, such as reduction, gather, scatter, and masked operations, are frequently encountered in layers like pooling, embedding lookups, and attention mechanisms. Understanding this terminology is essential for interpreting how low-level vector hardware accelerates high-level machine learning workloads.

Table 11.2: Vector operations and their neural network applications.

Vector Operation	Description	Neural Network Application
Reduction	Combines elements across a vector (e.g., sum, max)	Pooling layers, attention score computation
Gather	Loads multiple non-consecutive memory elements	Embedding lookups, sparse operations
Scatter	Writes to multiple non-consecutive memory locations	Gradient updates for embeddings
Masked operations	Selectively operates on vector elements	Attention masks, padding handling
Vector-scalar broadcast	Applies scalar to all vector elements	Bias addition, scaling operations

The efficiency gains from vector processing extend beyond instruction count reduction. Memory bandwidth utilization improves as vector loads transfer multiple values per operation. Energy efficiency increases because control logic is shared across multiple operations. These improvements compound across the deep layers of modern neural networks, where billions of operations execute for each forward pass.

11.3.1.4 Vector Processing History

The principles underlying vector operations have long played a central role in high-performance computing. In the 1970s and 1980s, vector processors emerged as a critical architectural solution for scientific computing, weather modeling, and physics simulations, where large arrays of data required efficient parallel processing. Early systems such as the Cray-1, one of the first commercially successful supercomputers, introduced dedicated vector units to perform arithmetic operations on entire data vectors in a single instruction. This approach dramatically improved computational throughput compared to traditional scalar execution (Jordan 1982).

These foundational concepts have reemerged in the context of machine learning, where neural networks exhibit an inherent structure well suited to vectorized execution. The same fundamental operations, such as vector addition, multiplication, and reduction, that once accelerated numerical simulations now drive the execution of machine learning workloads. While the scale and specialization of modern AI accelerators differ from their historical predecessors, the underlying architectural principles remain the same. The resurgence of vector processing in neural network acceleration highlights its enduring utility as a mechanism for achieving high computational efficiency.

Vector operations establish the foundation for neural network acceleration by enabling efficient parallel processing of independent data elements. However, the core transformations in neural networks require coordinating computation across multiple dimensions simultaneously. This need for structured parallel computation leads to the next architectural primitive: matrix operations.

11.3.2 Matrix Operations

Matrix operations are the computational workhorse of neural networks, transforming high-dimensional data through structured patterns of weights, activations, and gradients (I. J. Goodfellow, Courville, and Bengio 2013b). While vector operations process elements independently, matrix operations orchestrate computations across multiple dimensions simultaneously. Understanding these operations reveals fundamental patterns that drive hardware acceleration strategies.

11.3.2.1 Matrix Operations in NNs

Neural network computations decompose into hierarchical matrix operations. As shown in Listing 11.8, a linear layer demonstrates this hierarchy by transforming input features into output neurons over a batch.

This computation demonstrates the inherent scale of matrix operations in neural networks. Each output neuron (512 total) must process all input features (256 total) for every sample in the batch (32 samples). The weight matrix alone

Listing 11.8: Framework Level: What ML developers write

```
layer = nn.Linear(256, 512)  # Layer transforms 256 inputs to
                             # 512 outputs
output = layer(input_batch)  # Process a batch of 32 samples
# Framework Internal: Core operations
Z = matmul(weights, input)   # Matrix: transforms [256 x 32]
                             # input to [512 x 32] output
Z = Z + bias # Vector: adds bias to each output independently
output = relu(Z)             # Vector: applies activation to
                             # each element independently
```

contains $256 \times 512 = 131,072$ parameters that define these transformations, illustrating why efficient matrix multiplication becomes crucial for performance.

11.3.2.2 Matrix Computation Types in NNs

Matrix operations appear consistently across modern neural architectures, as illustrated in Listing 11.9.

Listing 11.9: Linear Layers – Direct matrix multiply

```
hidden = matmul(weights, inputs)
# weights: [out_dim x in_dim], inputs: [in_dim x batch]
# Result combines all inputs for each output
# Attention Mechanisms - Multiple matrix operations
Q = matmul(Wq, inputs)
# Project inputs to query space [query_dim x batch]
K = matmul(Wk, inputs)
# Project inputs to key space[key_dim x batch]
attention = matmul(Q, K.T)
# Compare all queries with all keys [query_dim x key_dim]
# Convolutions - Matrix multiply after reshaping
patches = im2col(input)
# Convert [H x W x C] image to matrix of patches
output = matmul(kernel, patches)
# Apply kernels to all patches simultaneously
```

This pervasive pattern of matrix multiplication has direct implications for hardware design. The need for efficient matrix operations drives the development of specialized hardware architectures that can handle these computations at scale. The following sections explore how modern AI accelerators implement matrix operations, focusing on their architectural features and performance optimizations.

11.3.2.3 Matrix Operations Hardware Acceleration

The computational demands of matrix operations have driven specialized hardware optimizations. Modern processors implement dedicated matrix units that extend beyond vector processing capabilities. An example of such matrix acceleration is shown in Listing 11.10.

Listing 11.10: Matrix unit operation for block-wise computation in hardware

```
mload mr1, (weight_ptr)      # Load e.g., 16x16 block of
                             # weight matrix
mload mr2, (input_ptr)       # Load corresponding input block
matmul.mm mr3, mr1, mr2      # Multiply and accumulate entire
                             # blocks at once
mstore (output_ptr), mr3     # Store computed output block
```

This matrix processing unit can handle 16×16 blocks of the linear layer computation described earlier, processing 256 multiply-accumulate operations simultaneously compared to the 8 operations possible with vector processing. These matrix operations complement vectorized computation by enabling structured many-to-many transformations. The interplay between matrix and vector operations shapes the efficiency of neural network execution.

Table 11.3: Comparison of matrix and vector operation characteristics.

Operation Type	Best For	Examples	Key Characteristic
Matrix Operations	Many-to-many transforms	Layer transformations Attention computation Convolutions	Each output depends on multiple inputs
Vector Operations	One-to-one transforms	Activation functions Layer normalization Element-wise gradients	Each output depends only on corresponding input

Matrix operations provide essential computational capabilities for neural networks through coordinated parallel processing across multiple dimensions (see Table 11.3). While they enable transformations such as attention mechanisms and convolutions, their performance depends on efficient data handling. Conversely, vector operations are optimized for one-to-one transformations like activation functions and layer normalization. The distinction between these operations highlights the importance of dataflow patterns in neural accelerator design, which we examine next (Hwu 2011).

11.3.2.4 Historical Foundations of Matrix Computation

Matrix operations have long served as a cornerstone of computational mathematics, with applications extending from numerical simulations to graphics processing (Golub and Loan 1996). The structured nature of matrix multiplications and transformations made them a natural target for acceleration in early computing architectures. In the 1980s and 1990s, specialized digital signal processors (DSPs) and graphics processing units (GPUs) optimized for matrix computations played a critical role in accelerating workloads such as image processing, scientific computing, and 3D rendering (Owens et al. 2008).

The widespread adoption of machine learning has reinforced the importance of efficient matrix computation. Neural networks, fundamentally built on matrix multiplications and tensor operations, have driven the development of dedicated hardware architectures that extend beyond traditional vector processing. Modern tensor processing units (TPUs) and AI accelerators implement matrix multiplication at scale, reflecting the same architectural principles that once underpinned early scientific computing and graphics workloads. The resurgence of matrix-centric architectures highlights the deep connection between classical numerical computing and contemporary AI acceleration.

11.3.3 Special Function Units

While vector and matrix operations efficiently handle the linear transformations in neural networks, non-linear functions present unique computational challenges that require dedicated hardware solutions. Special Function Units (SFUs) provide hardware acceleration for these essential computations, completing the set of fundamental processing primitives needed for efficient neural network execution.

11.3.3.1 Non-Linear Functions

Non-linear functions play a fundamental role in machine learning by enabling neural networks to model complex relationships (I. J. Goodfellow, Courville, and Bengio 2013c). Listing 11.11 illustrates a typical neural network layer sequence.

Listing 11.11: Typical layer sequence with non-linear operations

```
layer = nn.Sequential(
    nn.Linear(256, 512),
    nn.ReLU(),
    nn.BatchNorm1d(512)
)
output = layer(input_tensor)
```

This sequence introduces multiple non-linear transformations. As shown in Listing 11.12, the framework decomposes it into mathematical operations.

11.3.3.2 Non-Linear Functions Implementation

On traditional processors, these seemingly simple mathematical operations translate into complex sequences of instructions. Consider the computation of batch normalization: calculating the square root requires multiple iterations of numerical approximation, while exponential functions in operations like softmax need series expansion or lookup tables (Ioffe and Szegedy 2015b). Even

Listing 11.12: Mathematical operations from non-linear layer sequence

```
Z = matmul(weights, input) + bias    # Linear transformation
H = max(0, Z)                        # ReLU activation
mean = reduce_mean(H, axis=0)        # BatchNorm statistics
var = reduce_mean((H - mean)**2)     # Variance computation
output = gamma * (H - mean)/sqrt(var + eps) + beta
                                     # Normalization
```

a simple ReLU activation introduces branching logic that can disrupt instruction pipelining—see Listing 11.13 for an example.

Listing 11.13: Traditional implementation overhead for ReLU and BatchNorm

```
for batch in range(32):
    for feature in range(512):
       # ReLU: Requires branch prediction and potential
       # pipeline stalls
       z = matmul_output[batch, feature]
       h = max(0.0, z)    # Conditional operation

       # BatchNorm: Multiple passes over data
       mean_sum[feature] += h    # First pass for mean
       var_sum[feature] += h * h # Additional pass for variance
       temp[batch, feature] = h  # Extra memory storage needed

# Normalization requires complex arithmetic
for feature in range(512):
    mean = mean_sum[feature] / batch_size
    var = (var_sum[feature] / batch_size) - mean * mean
    # Square root computation: Multiple iterations
    scale = gamma[feature] / sqrt(var + eps)
    # Iterative approximation
    shift = beta[feature] - mean * scale
    # Additional pass over data for final computation
    for batch in range(32):
        output[batch, feature] = temp[batch, feature] *
                                 scale + shift
```

These operations introduce several key inefficiencies:

1. Multiple passes over data, increasing memory bandwidth requirements
2. Complex arithmetic requiring many instruction cycles
3. Conditional operations that can cause pipeline stalls
4. Additional memory storage for intermediate results

5. Poor utilization of vector processing units

More specifically, each operation introduces distinct challenges. Batch normalization requires multiple passes through data: one for mean computation, another for variance, and a final pass for output transformation. Each pass loads and stores data through the memory hierarchy. Operations that appear simple in mathematical notation often expand into many instructions. The square root computation typically requires 10-20 iterations of numerical methods like Newton-Raphson approximation for suitable precision (Goldberg 1991). Conditional operations like ReLU's max function require branch instructions that can stall the processor's pipeline. The implementation needs temporary storage for intermediate values, increasing memory usage and bandwidth consumption. While vector units excel at regular computations, functions like exponentials and square roots often require scalar operations that cannot fully utilize vector processing capabilities.

11.3.3.3 Hardware Acceleration

SFUs address these inefficiencies through dedicated hardware implementation. Modern ML accelerators include specialized circuits that transform these complex operations into single-cycle or fixed-latency computations. The accelerator can load a vector of values and apply non-linear functions directly, eliminating the need for multiple passes and complex instruction sequences as shown in Listing 11.14.

Listing 11.14: Hardware-accelerated non-linear vector operations

```
vld.v v1, (input_ptr)    # Load vector of values
vrelu.v v2, v1           # Single-cycle ReLU on entire vector
vsigm.v v3, v1           # Fixed-latency sigmoid computation
vtanh.v v4, v1           # Direct hardware tanh implementation
vrsqrt.v v5, v1          # Fast reciprocal square root
```

Each SFU implements a specific function through specialized circuitry. For instance, a ReLU unit performs the comparison and selection in dedicated logic, eliminating branching overhead. Square root operations use hardware implementations of algorithms like Newton-Raphson with fixed iteration counts, providing guaranteed latency. Exponential and logarithmic functions often combine small lookup tables with hardware interpolation circuits (Costa et al. 2019). Using these custom instructions, the SFU implementation eliminates multiple passes over data, removes complex arithmetic sequences, and maintains high computational efficiency. Table 11.4 shows the various hardware implementations and their typical latencies.

Table 11.4: Special function unit implementation.

Function Unit	Operation	Implementation Strategy	Typical Latency
Activation Unit	ReLU, sigmoid, tanh	Piece-wise approximation circuits	1-2 cycles
Statistics Unit	Mean, variance	Parallel reduction trees	log(N) cycles
Exponential Unit	exp, log	Table lookup + hardware interpolation	2-4 cycles
Root/Power Unit	sqrt, rsqrt	Fixed-iteration Newton-Raphson	4-8 cycles

11.3.3.4 SFUs History

The need for efficient non-linear function evaluation has shaped computer architecture for decades. Early processors incorporated hardware support for complex mathematical functions, such as logarithms and trigonometric operations, to accelerate workloads in scientific computing and signal processing (Smith 1997). In the 1970s and 1980s, floating-point co-processors were introduced to handle complex mathematical operations separately from the main CPU (Palmer 1980). In the 1990s, instruction set extensions such as Intel's SSE and ARM's NEON provided dedicated hardware for vectorized mathematical transformations, improving efficiency for multimedia and signal processing applications.

Machine learning workloads have reintroduced a strong demand for specialized functional units, as activation functions, normalization layers, and exponential transformations are fundamental to neural network computations. Rather than relying on iterative software approximations, modern AI accelerators implement fast, fixed-latency SFUs for these operations, mirroring historical trends in scientific computing. The reemergence of dedicated special function units underscores the ongoing cycle in hardware evolution, where domain-specific requirements drive the reinvention of classical architectural concepts in new computational paradigms.

The combination of vector, matrix, and special function units provides the computational foundation for modern AI accelerators. However, the effective utilization of these processing primitives depends critically on data movement and access patterns. This leads us to examine the architectures, hierarchies, and strategies that enable efficient data flow in neural network execution.

11.3.4 Compute Units and Execution Models

The vector operations, matrix operations, and special function units examined previously represent the fundamental computational primitives in AI accelerators. Modern AI processors package these primitives into distinct execution units, such as SIMD units, tensor cores, and processing elements, which define how computations are structured and exposed to users. Understanding this organization reveals both the theoretical capabilities and practical performance characteristics that developers can leverage in contemporary AI accelerators.

11.3.4.1 Primitive-Execution Unit Mapping

The progression from computational primitives to execution units follows a structured hierarchy that reflects the increasing complexity and specialization of AI accelerators:

- Vector operations → SIMD/SIMT units that enable parallel processing of independent data elements
- Matrix operations → Tensor cores and systolic arrays that provide structured matrix multiplication
- Special functions → Dedicated hardware units integrated within processing elements

Each execution unit combines these computational primitives with specialized memory and control mechanisms, optimizing both performance and energy efficiency. This structured packaging allows hardware vendors to expose standardized programming interfaces while implementing diverse underlying architectures tailored to specific workload requirements. The choice of execution unit significantly influences overall system efficiency, affecting data locality, compute density, and workload adaptability. Subsequent sections examine how these execution units operate within AI accelerators to maximize performance across different machine learning tasks.

11.3.4.2 SIMD to SIMT Transition

Single Instruction Multiple Data (SIMD) execution applies identical operations to multiple data elements in parallel, minimizing instruction overhead while maximizing data throughput. This execution model is widely used to accelerate workloads with regular, independent data parallelism, such as neural network computations. The ARM Scalable Vector Extension (SVE) provides a representative example of how modern architectures implement SIMD operations efficiently, as illustrated in Listing 11.15.

Listing 11.15: Vector operation implementation using ARM SVE

```
ptrue p0.s              # Create predicate for vector length
ld1w z0.s, p0/z, [x0]   # Load vector of inputs
fmul z1.s, z0.s, z0.s   # Multiply elements
fadd z2.s, z1.s, z0.s   # Add elements
st1w z2.s, p0, [x1]     # Store results
```

Processor architectures continue to expand SIMD capabilities to accommodate increasing computational demands. Intel's Advanced Matrix Extensions (AMX) and ARM's SVE2 architecture provide flexible SIMD execution, enabling software to scale across different hardware implementations (Stephens et al. 2017).

To address these limitations, SIMT extends SIMD principles by enabling parallel execution across multiple independent threads, each maintaining its own program counter and architectural state (E. Lindholm et al. 2008). This model maps naturally to matrix computations, where each thread processes different portions of a workload while still benefiting from shared instruction execution. In NVIDIA's GPU architectures, each Streaming Multiprocessor (SM) coordinates thousands of threads executing in parallel, allowing for efficient scaling of neural network computations, as demonstrated in Listing 11.16.

Listing 11.16: CUDA kernel for SIMT execution

```
__global__ void matrix_multiply(float* C, float* A, float*
                                B, int N) {
    // Each thread processes one output element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for (int k = 0; k < N; k++) {
        // Threads in a warp execute in parallel
        sum += A[row * N + k] * B[k * N + col];
    }
    C[row * N + col] = sum;
}
```

SIMT execution allows neural network computations to scale efficiently across thousands of threads while maintaining flexibility for divergent execution paths. Similar execution models appear in AMD's RDNA and Intel's Xe architectures, reinforcing SIMT as a fundamental mechanism for AI acceleration.

11.3.4.3 Tensor Cores

While SIMD and SIMT units provide efficient execution of vector operations, neural networks rely heavily on matrix computations that require specialized execution units for structured multi-dimensional processing. Tensor processing units extend SIMD and SIMT principles by enabling efficient matrix operations through dedicated hardware blocks. These units execute matrix multiplications and accumulations on entire matrix blocks in a single operation, reducing instruction overhead and optimizing data movement.

Tensor cores, implemented in architectures such as NVIDIA's Ampere GPUs, provide an example of this approach. They expose matrix computation capabilities through specialized instructions, such as the tensor core operation shown in Listing 11.17 on the NVIDIA A100 GPU.

Listing 11.17: Tensor Core operation on NVIDIA A100 GPU

```
Tensor Core Operation (NVIDIA A100):
mma.sync.aligned.m16n16k16.f16.f16
  {d0,d1,d2,d3},      // Destination registers
  {a0,a1,a2,a3},      // Source matrix A
  {b0,b1,b2,b3},      // Source matrix B
  {c0,c1,c2,c3}       // Accumulator
```

A single tensor core instruction processes an entire matrix block while maintaining intermediate results in local registers, significantly improving computational efficiency compared to implementations based on scalar or vector operations. This structured approach enables hardware to achieve high throughput while reducing the burden of explicit loop unrolling and data management at the software level.

Tensor processing unit architectures differ based on design priorities. NVIDIA's Ampere architecture incorporates tensor cores optimized for general-purpose deep learning acceleration. Google's TPUv4 utilizes large-scale matrix units arranged in systolic arrays to maximize sustained training throughput. Apple's M1 neural engine integrates smaller matrix processors optimized for mobile inference workloads, while Intel's Sapphire Rapids architecture introduces AMX tiles designed for high-performance datacenter applications.

The increasing specialization of AI hardware has driven significant performance improvements in deep learning workloads. Figure 11.3 illustrates the trajectory of AI accelerator performance in NVIDIA GPUs, highlighting the transition from general-purpose floating-point execution units to highly optimized tensor processing cores.

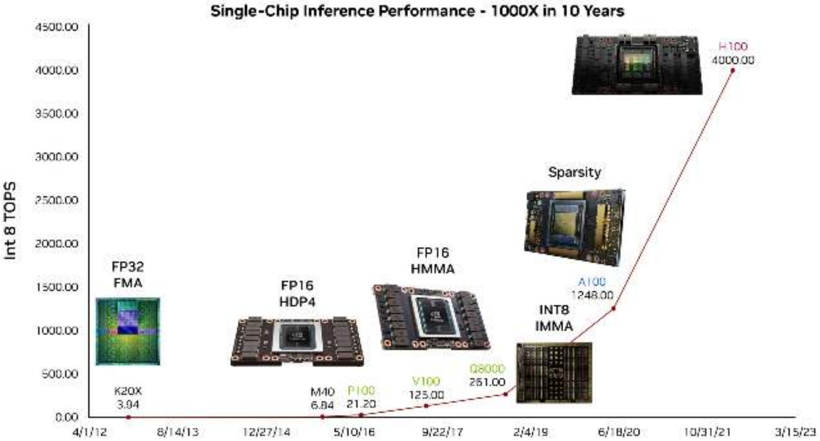


Figure 11.3: Single-chip performance scaling.

11.3.4.4 Processing Elements

The highest level of execution unit organization integrates multiple tensor cores with local memory into processing elements (PEs). A processing element serves as a fundamental building block in many AI accelerators, combining different computational units to efficiently execute neural network operations. Each PE typically includes vector units for element-wise operations, tensor cores for matrix computation, special function units for non-linear transformations, and dedicated memory resources to optimize data locality and minimize data movement overhead.

Processing elements play an essential role in AI hardware by balancing computational density with memory access efficiency. Their design varies across different architectures to support diverse workloads and scalability requirements. Graphcore's Intelligence Processing Unit (IPU) distributes computation across 1,472 tiles, each containing independent processing elements optimized for fine-grained parallelism (Graphcore 2020). Cerebras extends this approach in the CS-2 system, integrating 850,000 processing elements across a wafer-scale device to accelerate sparse computations. Tesla's D1 processor arranges processing elements with substantial local memory, optimizing throughput and latency for real-time autonomous vehicle workloads (T. Inc. 2021).

Processing elements provide the structural foundation for large-scale AI acceleration. Their efficiency depends not only on computational capability but also on interconnect strategies and memory hierarchy design. The next sections explore how these architectural choices impact performance across different AI workloads.

Tensor processing units have enabled substantial efficiency gains in AI workloads by leveraging hardware-accelerated matrix computation. Their role continues to evolve as architectures incorporate support for advanced execution techniques, including structured sparsity[4] and workload-specific optimizations. The effectiveness of these units, however, depends not only on their computational capabilities but also on how they interact with memory hierarchies and data movement mechanisms, which are examined in subsequent sections.

11.3.4.5 Systolic Arrays

While tensor cores package matrix operations into structured computational units, systolic arrays provide an alternative approach optimized for continuous data flow and operand reuse. A systolic array arranges processing elements in a grid pattern, where data flows rhythmically between neighboring units in a synchronized manner. This structured movement of data enables efficient execution of matrix multiplication, reducing memory access overhead and maximizing computational throughput.

The concept of systolic arrays was first introduced by H.T. Kung, who formalized their use in parallel computing architectures for efficient matrix operations (Kung 1982). Unlike general-purpose execution units, systolic arrays exploit spatial and temporal locality by reusing operands as they propagate through the grid. Google's Tensor Processing Unit (TPU) exemplifies this architectural approach. In the TPUv4, a 128×128 systolic array of multiply-accumulate units processes matrix operations by streaming data through the array in a pipelined manner, as shown in Figure 11.4.

Each processing element in the array performs a multiply-accumulate operation in every cycle:

1. Receives an input activation from above
2. Receives a weight value from the left
3. Multiplies these values and adds to its running sum
4. Passes the input activation downward and the weight value rightward to neighboring elements

This structured computation model minimizes data movement between global memory and processing elements, improving both efficiency and scalability. As systolic arrays operate in a streaming fashion, they are particularly

[4] Structured Sparsity: The deliberate design of neural network weight matrices where entire rows, columns, or blocks are pruned, thus simplifying hardware implementation and improving efficiency.

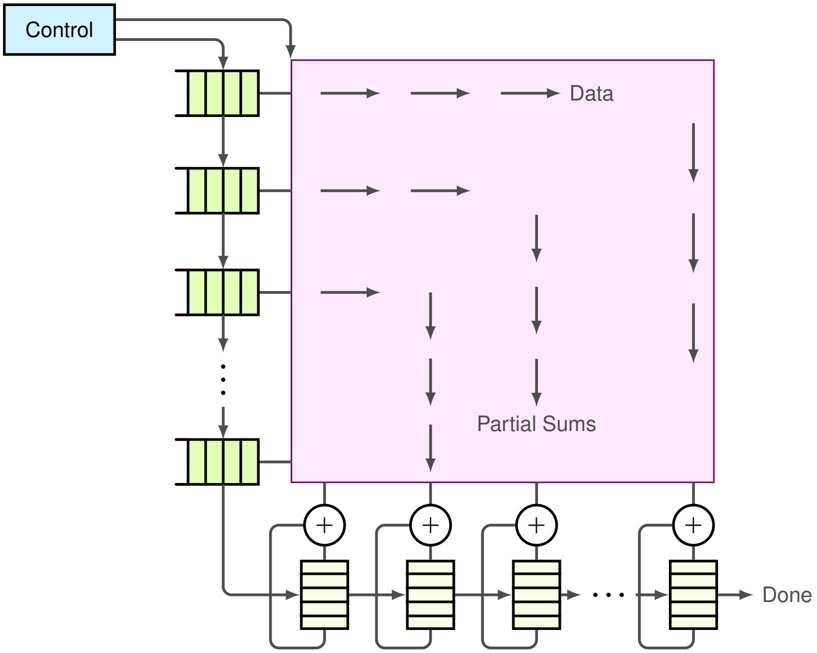


Figure 11.4: Data flow movement in a systolic array.

effective for high-throughput workloads such as deep learning training and inference.

While the diagram in Figure 11.4 illustrates one common systolic array implementation, systolic architectures vary significantly across different accelerator designs. Training-focused architectures like Google's TPU employ large arrays optimized for high computational throughput, while inference-oriented designs found in edge devices prioritize energy efficiency with smaller configurations.

The fundamental principle remains consistent: data flows systematically through processing elements, with inputs moving horizontally and vertically to compute partial sums in a synchronized fashion. However, the practical effectiveness of systolic arrays extends beyond their computational structure—it depends heavily on efficient memory access patterns and careful scheduling strategies, topics we explore in detail in subsequent sections.

11.3.4.6 Numerics in AI Acceleration

The efficiency of AI accelerators is not determined by computational power alone but also by the precision of numerical representations. The choice of numerical format shapes the balance between accuracy, throughput, and energy consumption, influencing how different execution units, such as SIMD and SIMT units, tensor cores, and systolic arrays, are designed and deployed.

Precision Trade-offs. Numerical precision represents a critical design parameter in modern AI accelerators. While higher precision formats provide mathematical stability and accuracy, they come with substantial costs in terms of power consumption, memory bandwidth, and computational throughput. Finding the optimal precision point has become a central challenge in AI hardware architecture.

Early deep learning models primarily relied on single-precision floating point (FP32) for both training and inference. While FP32 offers sufficient dynamic range and precision for stable learning, it imposes high computational and memory costs, limiting efficiency, especially as model sizes increase. Over time, hardware architectures evolved to support lower precision formats such as half-precision floating point (FP16) and bfloat16 (BF16), which reduce memory usage and increase computational throughput while maintaining sufficient accuracy for deep learning tasks. More recently, integer formats (INT8, INT4) have gained prominence in inference workloads, where small numerical representations significantly improve energy efficiency without compromising model accuracy beyond acceptable limits.

The transition from high-precision to lower-precision formats is deeply integrated into hardware execution models. SIMD and SIMT units provide flexible support for multiple precisions, dynamically adapting to workload requirements. Tensor cores are designed explicitly for matrix multiplications, accelerating computation using reduced-precision floating point and integer arithmetic. Systolic arrays, with their structured data flow, further optimize performance by minimizing memory bandwidth constraints, often favoring low-precision formats that maximize operand reuse.

Despite the advantages of reduced precision, deep learning models cannot always rely solely on low-bit representations. To address this challenge, modern AI accelerators implement mixed-precision computing, where different numerical formats are used at different stages of execution. For example, matrix multiplications may be performed in FP16 or BF16, while accumulations are maintained in FP32 to prevent precision loss. Similarly, inference engines leverage INT8 arithmetic while preserving key activations in higher precision when necessary.

Mixed-Precision Computing. Modern AI accelerators increasingly support mixed-precision execution, allowing different numerical formats to be used at various stages of computation. Training workloads often leverage FP16 or BF16 for matrix multiplications, while maintaining FP32 accumulations to preserve precision. Inference workloads, by contrast, optimize for INT8 or even INT4, achieving high efficiency while retaining acceptable accuracy.

This shift toward precision diversity is evident in the evolution of AI hardware. Early architectures such as NVIDIA Volta provided limited support for lower precision beyond FP16, whereas later architectures, including Turing and Ampere, expanded the range of supported formats. Ampere GPUs introduced TF32 as a hybrid between FP32 and FP16, alongside broader support for BF16, INT8, and INT4. Table 11.5 illustrates this trend.

Table 11.5: Tensor core and CUDA core precisions across GPU architectures.

Architecture	Year	Supported Tensor Core Precisions	Supported CUDA Core Precisions
Volta	2017	FP16	FP64, FP32, FP16

Architecture	Year	Supported Tensor Core Precisions	Supported CUDA Core Precisions
Turing	2018	FP16, INT8	FP64, FP32, FP16, INT8
Ampere	2020	FP64, TF32, bfloat16, FP16, INT8, INT4	FP64, FP32, FP16, bfloat16, INT8

Table 11.5 highlights how newer architectures incorporate a growing diversity of numerical formats, reflecting the need for greater flexibility across different AI workloads. This trend suggests that future AI accelerators will continue expanding support for adaptive precision, optimizing both computational efficiency and model accuracy. The selection now reads:

The precision format used in hardware design has far-reaching implications. By adopting lower-precision formats, the data transferred between execution units and memory is reduced, leading to decreased memory bandwidth requirements and storage. Moreover, tensor cores and systolic arrays can process more lower-precision elements in parallel, thereby increasing the effective throughput in terms of FLOPs. Energy efficiency is also improved, as integer-based computations (e.g., INT8) require lower power compared to floating-point arithmetic—a clear advantage for inference workloads.

As AI models continue to scale in size, accelerator architectures are evolving to support more efficient numerical formats. Future designs are expected to incorporate adaptive precision techniques, dynamically adjusting computation precision based on workload characteristics. This evolution promises further optimization of deep learning performance while striking an optimal balance between accuracy and energy efficiency.

11.3.4.7 Architectural Integration

The organization of computational primitives into execution units determines the efficiency of AI accelerators. While SIMD, tensor cores, and systolic arrays serve as fundamental building blocks, their integration into full-chip architectures varies significantly across different AI processors. The choice of execution units, their numerical precision support, and their connectivity impact how effectively hardware can scale for deep learning workloads.

Modern AI processors exhibit a range of design trade-offs based on their intended applications. Some architectures, such as NVIDIA's A100, integrate large numbers of tensor cores optimized for FP16-based training, while Google's TPUv4 prioritizes high-throughput BF16 matrix multiplications. Inference-focused processors, such as Intel's Sapphire Rapids, incorporate INT8-optimized tensor cores to maximize efficiency. The Apple M1, designed for mobile workloads, employs smaller processing elements optimized for low-power FP16 execution. These design choices reflect the growing flexibility in numerical precision and execution unit organization, as discussed in the previous section.

Table 11.6 summarizes the execution unit configurations across contemporary AI processors.

Table 11.6: Execution unit configurations across modern AI processors

Processor	SIMD Width	Tensor Core Size	Processing Elements	Primary Workloads
NVIDIA A100	1024-bit	$4 \times 4 \times 4$ FP16	108 SMs	Training, HPC
Google TPUv4	128-wide	128×128 BF16	2 cores/chip	Training
Intel Sapphire	512-bit AVX	32×32 INT8/BF16	56 cores	Inference
Apple M1	128-bit NEON	16×16 FP16	8 NPU cores	Mobile inference

Table 11.6 highlights how execution unit configurations vary across architectures to optimize for different deep learning workloads. Training accelerators prioritize high-throughput floating-point tensor operations, whereas inference processors focus on low-precision integer execution for efficiency. Meanwhile, mobile accelerators balance precision and power efficiency to meet real-time constraints.

While execution units define the compute potential of an accelerator, their effectiveness is fundamentally constrained by data movement and memory hierarchy. Achieving high utilization of compute resources requires efficient memory systems that minimize data transfer overhead and optimize locality. The next section explores these architectural challenges, focusing on how memory hierarchy impacts AI accelerator performance.

11.4 AI Memory Systems

Machine learning accelerators are designed to maximize computational throughput, leveraging specialized primitives such as vector units, matrix engines, and systolic arrays. However, the efficiency of these compute units is fundamentally constrained by the availability of data. Unlike conventional workloads, ML models require frequent access to large volumes of parameters, activations, and intermediate results, leading to substantial memory bandwidth demands. If data cannot be delivered to the processing elements at the required rate, memory bottlenecks can significantly limit performance, regardless of the accelerator's raw computational capability.

Modern AI hardware leverages advanced memory hierarchies, efficient data movement techniques, and compression strategies to alleviate bottlenecks and enhance performance. By examining the interplay between ML workloads and memory systems along with memory bandwidth constraints, we can gain insights into architectural innovations that promote efficient execution and improved AI acceleration.

11.4.1 AI Memory Wall

Machine learning accelerators are capable of performing vast amounts of computation per cycle, but their efficiency is increasingly limited by data movement rather than raw processing power. The disparity between rapid computational advancements and slower memory performance has led to a growing bottleneck, often referred to as the AI memory wall. Even the most optimized hardware architectures struggle to sustain peak throughput if data cannot be delivered at the required rate. Ensuring that compute units remain fully utilized without being stalled by memory latency and bandwidth constraints is one of the central challenges in AI acceleration.

11.4.1.1 Compute-Memory Imbalance

As we have seen, neural networks rely on specialized computational primitives such as vector operations, matrix multiplications, and domain-specific functional units that accelerate key aspects of machine learning workloads. These operations are designed for highly parallel execution, enabling accelerators to perform vast amounts of computation in each cycle. Given this level of specialization, one might expect neural networks to execute efficiently without significant bottlenecks. However, the primary constraint is not the raw compute power but rather the ability to continuously supply data to these processing units.

While these compute units can execute millions of operations per second, they remain heavily dependent on memory bandwidth to sustain peak performance. Each matrix multiplication or vector operation requires a steady flow of weights, activations, and intermediate results, all of which must be fetched from memory. If data cannot be delivered at the required rate, memory stalls occur, leaving many compute units idle. This imbalance between computational capability and data availability is often referred to as the memory wall—a fundamental challenge in AI acceleration.

Over time, the gap between computation and memory performance has widened. As illustrated in Figure 11.5, the shaded region, referred to as the "AI Memory Wall," highlights the growing disparity between compute performance and memory bandwidth over time. This visualization underscores the compute-memory imbalance, where computational capabilities advance rapidly while memory bandwidth lags, leading to potential bottlenecks in data-intensive applications. Over the past 20 years, peak server hardware FLOPs have scaled at 3.0x every two years, far outpacing the growth of DRAM bandwidth (1.6x/2yrs) (A. Gholami et al. 2024). This growing imbalance has made memory bandwidth, rather than compute, the primary constraint in AI acceleration.

Beyond performance limitations, memory access imposes a significant energy cost[5]. Fetching data from off-chip DRAM, in particular, consumes far more energy than performing arithmetic operations (Horowitz 2014). This inefficiency is particularly evident in machine learning models, where large parameter sizes, frequent memory accesses, and non-uniform data movement patterns exacerbate memory bottlenecks.

[5] A 32-bit floating-point addition consumes approximately 20 fJ, while fetching two 32-bit words from off-chip DRAM costs around 1.3 nJ–a difference of 65,000.

11.4.1.2 Memory-Intensive ML Workloads

Machine learning workloads place substantial demands on memory systems due to the large volume of data involved in computation. Unlike traditional compute-bound applications, where performance is often dictated by the speed of arithmetic operations, ML workloads are characterized by high data movement requirements. The efficiency of an accelerator is not solely determined by its computational throughput but also by its ability to continuously supply data to processing units without introducing stalls or delays.

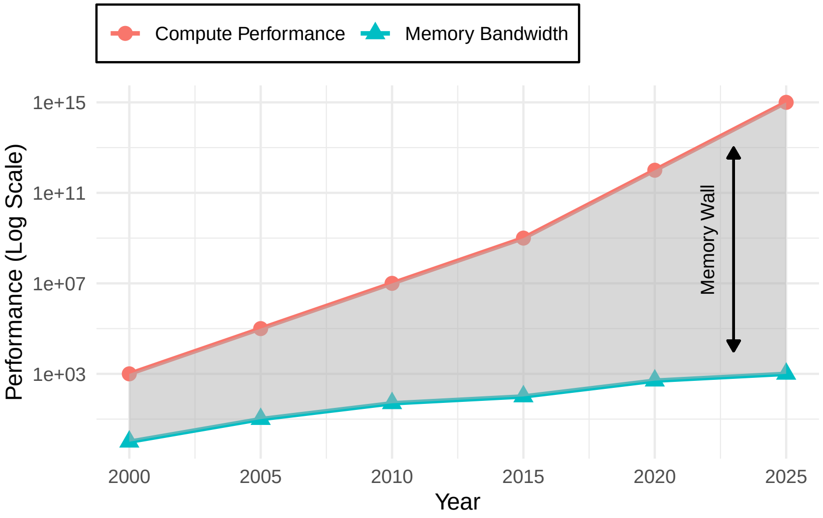


Figure 11.5: Compute performance versus memory bandwidth over time.

A neural network processes multiple types of data throughout its execution, each with distinct memory access patterns:

- **Model parameters (weights and biases)**: Machine learning models, particularly those used in large-scale applications such as natural language processing and computer vision, often contain millions to billions of parameters. Storing and accessing these weights efficiently is essential for maintaining throughput.
- **Intermediate activations**: During both training and inference, each layer produces intermediate results that must be temporarily stored and retrieved for subsequent operations. These activations can contribute significantly to memory overhead, particularly in deep architectures.
- **Gradients (during training)**: Backpropagation requires storing and accessing gradients for every parameter, further increasing the volume of data movement between compute units and memory.

As models increase in size and complexity, improvements in memory capacity and bandwidth become essential. Although specialized compute units accelerate operations like matrix multiplications, their overall performance depends on the continuous, efficient delivery of data to the processing elements. In large-scale applications, such as natural language processing and computer vision, models often incorporate millions to billions of parameters (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, Dhariwal, et al. 2020). Consequently, achieving high performance necessitates minimizing delays and stalls caused by inefficient data movement between memory and compute units (D. Narayanan et al. 2021a; Xingyu 2019).

One way to quantify this challenge is by comparing the data transfer time with the time required for computations. Specifically, we define the memory transfer time as

$$T_{\text{mem}} = \frac{M_{\text{total}}}{B_{\text{mem}}},$$

where M_{total} is the total data volume and B_{mem} is the available memory bandwidth. In contrast, the compute time is given by

$$T_{\text{compute}} = \frac{\text{FLOPs}}{P_{\text{peak}}},$$

with the number of floating-point operations (FLOPs) divided by the peak hardware throughput, P_{peak}. When $T_{\text{mem}} > T_{\text{compute}}$, the system becomes memory-bound, meaning that the processing elements spend more time waiting for data than performing computations. This imbalance demonstrates the need for memory-optimized architectures and efficient data movement strategies to sustain high performance.

Figure 11.6 demonstrates the emerging challenge between model growth and hardware memory capabilities, illustrating the "AI Memory Wall." The figure tracks AI model sizes (red dots) and hardware memory bandwidth (blue dots) over time on a log scale. Model parameters have grown exponentially, from AlexNet's modest 60M parameters in 2012 to Gemini 1's trillion-scale parameters in 2023, as shown by the steeper red trend line. In contrast, hardware memory bandwidth, represented by successive generations of NVIDIA GPUs (~100-200 GB/s) and Google TPUs (~2-3 TB/s), has increased more gradually (blue trend line). The expanding shaded region between these trends corresponds to the "AI Memory Wall," which will be an architectural challenge where model scaling outpaces available memory bandwidth. This growing disparity necessitates increasingly sophisticated memory management and model optimization techniques to maintain computational efficiency.

11.4.1.3 Irregular Memory Access

Unlike traditional computing workloads, where memory access follows well-structured and predictable patterns, machine learning models often exhibit irregular memory access behaviors that make efficient data retrieval a challenge. These irregularities arise due to the nature of ML computations, where memory access patterns are influenced by factors such as batch size, layer type, and sparsity. As a result, standard caching mechanisms and memory hierarchies often struggle to optimize performance, leading to increased memory latency and inefficient bandwidth utilization.

To better understand how ML workloads differ from traditional computing workloads, it is useful to compare their respective memory access patterns (Table 11.7). Traditional workloads, such as scientific computing, general-purpose CPU applications, and database processing, typically exhibit well-defined memory access characteristics that benefit from standard caching and prefetching techniques. ML workloads, on the other hand, introduce highly dynamic access patterns that challenge conventional memory optimization strategies.

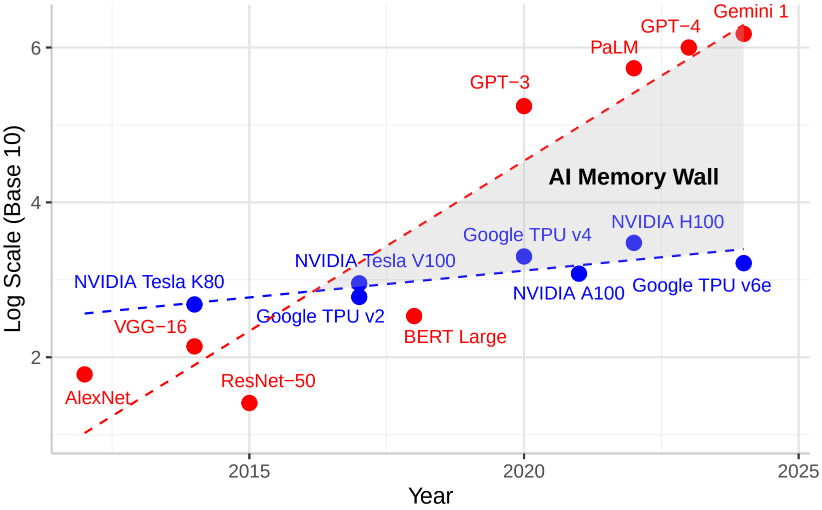


Figure 11.6: Model growth (in parameters) versus memory bandwidth (in GB/s).

Table 11.7: Memory access patterns in traditional vs. ML workloads.

Feature	Traditional Computing Workloads	Machine Learning Workloads
Memory Access Pattern	Regular and predictable (e.g., sequential reads, structured patterns)	Irregular and dynamic (e.g., sparsity, attention mechanisms)
Cache Locality	High temporal and spatial locality	Often low locality, especially in large models
Data Reuse	Structured loops with frequent data reuse	Sparse and dynamic reuse depending on layer type
Data Dependencies	Well-defined dependencies allow efficient prefetching	Variable dependencies based on network structure
Workload Example	Scientific computing (e.g., matrix factorizations, physics simulations)	Neural networks (e.g., CNNs, Transformers, sparse models)
Memory Bottleneck	DRAM latency, cache misses	Off-chip bandwidth constraints, memory fragmentation
Impact on Energy Consumption	Moderate, driven by FLOP-heavy execution	High, dominated by data movement costs

One key source of irregularity in ML workloads stems from batch size and execution order. The way input data is processed in batches directly affects memory reuse, creating a complex optimization challenge. Small batch sizes decrease the likelihood of reusing cached activations and weights, resulting in frequent memory fetches from slower, off-chip memory. Larger batch sizes can improve reuse and amortize memory access costs, but simultaneously place higher demands on available memory bandwidth, potentially creating congestion at different memory hierarchy levels. This delicate balance requires careful consideration of model architecture and available hardware resources.

In addition to batch size, different neural network layers interact with memory in distinct ways. Convolutional layers benefit from spatial locality, as neighboring pixels in an image are processed together, allowing for efficient caching of small weight kernels. Conversely, fully connected layers require frequent access to large weight matrices, often leading to more randomized memory access patterns that poorly align with standard caching policies. Transformers introduce additional complexity, as attention mechanisms demand accessing large key-value pairs stored across varied memory locations. The dynamic nature of sequence length and attention span renders traditional prefetching strategies ineffective, resulting in unpredictable memory latencies.

Another significant factor contributing to irregular memory access is sparsity in neural networks. Many modern ML models employ techniques such as weight pruning, activation sparsity, and structured sparsity to reduce computational overhead. However, these optimizations often lead to non-uniform memory access, as sparse representations necessitate fetching scattered elements rather than sequential blocks, making hardware caching less effective. Furthermore, models that incorporate dynamic computation paths, such as Mixture of Experts[6] and Adaptive Computation Time[7], introduce highly non-deterministic memory access patterns, where the active neurons or model components can vary with each inference step. This variability challenges efficient prefetching and caching strategies.

The consequences of these irregularities are significant. ML workloads often experience reduced cache efficiency, as activations and weights may not be accessed in predictable sequences. This leads to increased reliance on off-chip memory traffic, which not only slows down execution but also consumes more energy. Additionally, irregular access patterns contribute to memory fragmentation, where the way data is allocated and retrieved results in inefficient utilization of available memory resources. The combined effect of these factors is that ML accelerators frequently encounter memory bottlenecks that limit their ability to fully utilize available compute power.

11.4.2 Memory Hierarchy

To address the memory challenges in ML acceleration, hardware designers implement sophisticated memory hierarchies that balance speed, capacity, and energy efficiency. Understanding this hierarchy is essential before examining how different ML architectures utilize memory resources. Unlike general-purpose computing, where memory access patterns are often unpredictable, ML workloads exhibit structured reuse patterns that can be optimized through careful organization of data across multiple memory levels.

Unlike general-purpose computing, where memory access patterns are often unpredictable, machine learning workloads exhibit structured reuse patterns that can be optimized by carefully organizing data across multiple levels of memory. At the highest level, large-capacity but slow storage devices provide long-term model storage. At the lowest level, high-speed registers and caches ensure that compute units can access operands with minimal latency. Between these extremes, intermediate memory levels, such as scratchpad memory, high-bandwidth memory, and off-chip DRAM, offer trade-offs between performance and capacity.

Table 11.8 summarizes the key characteristics of different memory levels in modern AI accelerators. Each level in the hierarchy has distinct latency, bandwidth, and capacity properties, which directly influence how neural network data, such as weights, activations, and intermediate results, should be allocated.

[6] Mixture of Experts: A model design where different inputs are routed to specialized subnetworks based on gating mechanisms.

[7] Adaptive Computation Time: Allowing a network to dynamically allocate varying amounts of computation to different inputs based on their complexity.

Table 11.8: Memory hierarchy characteristics and their impact on machine learning.

Memory Level	Approx. Latency	Band-width	Capacity	Example Use in Deep Learning
Registers	~1 cycle	Highest	Few values	Storing operands for immediate computation
L1/L2 Cache (SRAM)	~1-10 ns	High	KBs-MBs	Caching frequently accessed activations and small weight blocks
Scratchpad Memory	~5-20 ns	High	MBs	Software-managed storage for intermediate computations
High-Bandwidth Memory (HBM)	~100 ns	Very High	GBs	Storing large model parameters and activations for high-speed access
Off-Chip DRAM (DDR, GDDR, LPDDR)	~50-150 ns	Moderate	GBs-TBs	Storing entire model weights that do not fit on-chip
Flash Storage (SSD/NVMe)	~100 µs - 1 ms	Low	TBs	Storing pre-trained models and checkpoints for later loading

11.4.2.1 On-Chip Memory

Each level of the memory hierarchy serves a distinct role in AI acceleration, with different trade-offs in speed, capacity, and accessibility. Registers, located within compute cores, provide the fastest access but can only store a few operands at a time. These are best utilized for immediate computations, where the operands needed for an operation can be loaded and consumed within a few cycles. However, because register storage is so limited, frequent memory accesses are required to fetch new operands and store intermediate results.

To reduce the need for constant data movement between registers and external memory, small but fast caches serve as an intermediary buffer. These caches store recently accessed activations, weights, and intermediate values, ensuring that frequently used data remains available with minimal delay. However, the size of caches is limited, making them insufficient for storing full feature maps or large weight tensors in machine learning models. As a result, only the most frequently used portions of a model's parameters or activations can reside here at any given time.

For larger working datasets, many AI accelerators include scratchpad memory, which offers more storage than caches but with a crucial difference: it allows explicit software control over what data is stored and when it is evicted. Unlike caches, which rely on hardware-based eviction policies, scratchpad memory enables machine learning workloads to retain key values such as activations and filter weights for multiple layers of computation. This capability is particularly useful in models like convolutional neural networks, where the same input feature maps and filter weights are reused across multiple operations. By keeping this data in scratchpad memory rather than reloading it from external memory, accelerators can significantly reduce unnecessary memory transfers and improve overall efficiency (Y.-H. Chen, Emer, and Sze 2017).

11.4.2.2 Off-Chip Memory

Beyond on-chip memory, high-bandwidth memory provides rapid access to larger model parameters and activations that do not fit within caches or scratchpad buffers. HBM achieves its high performance by stacking multiple memory dies and using wide memory interfaces, allowing it to transfer large amounts of data with minimal latency compared to traditional DRAM. Because of its high bandwidth and lower latency, HBM is often used to store entire layers of machine learning models that must be accessed quickly during execution. However, its cost and power consumption limit its use primarily to high-performance AI accelerators, making it less common in power-constrained environments such as edge devices.

When a machine learning model exceeds the capacity of on-chip memory and HBM, it must rely on off-chip DRAM, such as DDR, GDDR, or LPDDR. While DRAM offers significantly greater storage capacity, its access latency is higher, meaning that frequent retrievals from DRAM can introduce execution bottlenecks. To make effective use of DRAM, models must be structured so that only the necessary portions of weights and activations are retrieved at any given time, minimizing the impact of long memory fetch times.

At the highest level of the hierarchy, flash storage and solid-state drives (SSDs) store large pre-trained models, datasets, and checkpointed weights. These storage devices offer large capacities but are too slow for real-time execution, requiring models to be loaded into faster memory tiers before computation begins. For instance, in training scenarios, checkpointed models stored in SSDs must be loaded into DRAM or HBM before resuming computation, as direct execution from SSDs would be too slow to maintain efficient accelerator utilization (D. Narayanan et al. 2021a).

The memory hierarchy balances competing objectives of speed, capacity, and energy efficiency. However, moving data through multiple memory levels introduces bottlenecks that limit accelerator performance. Data transfers between memory levels incur latency costs, particularly for off-chip accesses. Limited bandwidth restricts data flow between memory tiers. Memory capacity constraints force constant data movement as models exceed local storage.

11.4.3 Host-Accelerator Communication

Machine learning accelerators, such as GPUs and TPUs, achieve high computational throughput through parallel execution. However, their efficiency is fundamentally constrained by data movement between the host (CPU) and accelerator memory. Unlike general-purpose workloads that operate entirely within a CPU's memory subsystem, AI workloads require frequent data transfers between CPU main memory and the accelerator, introducing latency, consuming bandwidth, and affecting overall performance.

Host-accelerator data movement follows a structured sequence, as illustrated in Figure 11.7. Before computation begins, data is copied from CPU memory to the accelerator's memory. The CPU then issues execution instructions, and the accelerator processes the data in parallel. Once computation completes, the results are stored in accelerator memory and transferred back to the CPU. Each step introduces potential inefficiencies that must be managed to optimize performance.

The key challenges in host-accelerator data movement include latency, bandwidth constraints, and synchronization overheads. Optimizing data transfers

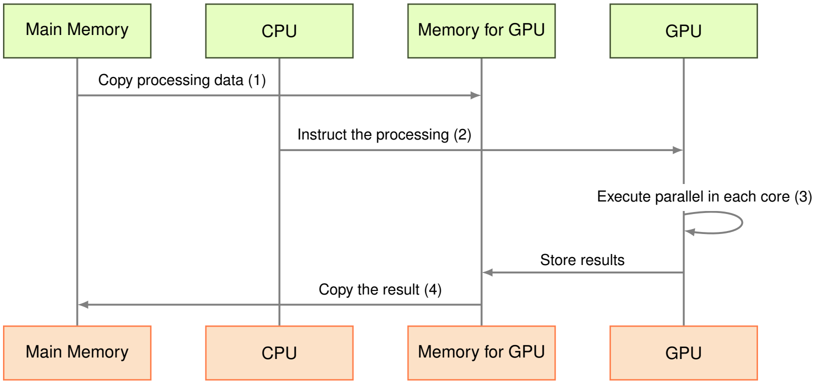


Figure 11.7: Host-accelerator memory access interactions.

through efficient memory management and interconnect technologies is essential for maximizing accelerator utilization.

11.4.3.1 Data Transfer Patterns

The efficiency of ML accelerators depends not only on their computational power but also on the continuous supply of data. Even high-performance GPUs and TPUs remain underutilized if data transfers are inefficient. Host and accelerator memory exist as separate domains, requiring explicit transfers over interconnects such as PCIe, NVLink, or proprietary links. Ineffective data movement can cause execution stalls, making transfer optimization critical.

Figure 11.7 illustrates this structured sequence. In step (1), data is copied from CPU memory to accelerator memory, as GPUs cannot directly access host memory at high speeds. A direct memory access (DMA) engine typically handles this transfer without consuming CPU cycles. In step (2), the CPU issues execution commands via APIs like CUDA, ROCm, or OpenCL. Step (3) involves parallel execution on the accelerator, where stalls can occur if data is not available when needed. Finally, in step (4), computed results are copied back to CPU memory for further processing.

Latency and bandwidth limitations significantly impact AI workloads. PCIe, with a peak bandwidth of 32 GB/s (PCIe 4.0), is much slower than an accelerator's high-bandwidth memory, which can exceed 1 TB/s. Large data transfers exacerbate bottlenecks, particularly in deep learning tasks. Additionally, synchronization overheads arise when computation must wait for data transfers to complete. Efficient scheduling and overlapping transfers with execution are essential to mitigate these inefficiencies.

11.4.3.2 Data Transfer Mechanisms

The movement of data between the host (CPU) and the accelerator (GPU, TPU, or other AI hardware) depends on the interconnect technology that links the two processing units. The choice of interconnect determines the bandwidth available for transfers, the latency of communication, and the overall efficiency of host-accelerator execution. The most commonly used transfer mechanisms include PCIe (Peripheral Component Interconnect Express), NVLink, Direct Memory Access, and Unified Memory Architectures. Each of these plays a crucial role in optimizing the four-step data movement process illustrated in Figure 11.7.

PCIe Interface. Most accelerators communicate with the CPU via PCIe, the industry-standard interconnect for data movement. PCIe 4.0 provides up to 32 GB/s bandwidth, while PCIe 5.0 doubles this to 64 GB/s. However, this is still significantly lower than HBM bandwidth within accelerators, making PCIe a bottleneck for large AI workloads.

PCIe also introduces latency overheads due to its packet-based communication and memory-mapped I/O model. Frequent small transfers are inefficient, so batching data movement reduces overhead. Computation commands, issued over PCIe, further contribute to latency, requiring careful optimization of execution scheduling.

NVLink Interface. To address the bandwidth limitations of PCIe, NVIDIA developed NVLink, a proprietary high-speed interconnect that provides significantly higher bandwidth between GPUs and, in some configurations, between the CPU and GPU. Unlike PCIe, which operates as a shared bus, NVLink enables direct point-to-point communication between connected devices, reducing contention and improving efficiency for AI workloads.

For host-accelerator transfers, NVLink can be used in step (1) to transfer input data from main memory to GPU memory at speeds far exceeding PCIe, with bandwidths reaching up to 600 GB/s in NVLink 4.0. This significantly reduces the data movement bottleneck, allowing accelerators to access input data with lower latency. In multi-GPU configurations, NVLink also accelerates peer-to-peer transfers, allowing accelerators to exchange data without routing through main memory, thereby optimizing step (3) of the computation process.

Although NVLink offers substantial performance benefits, it is not universally available. Unlike PCIe, which is an industry standard across all accelerators, NVLink is specific to NVIDIA hardware, limiting its applicability to systems designed with NVLink-enabled GPUs.

DMA for Data Transfers. In conventional memory transfers, the CPU issues load/store instructions, consuming processing cycles. DMA offloads this task, enabling asynchronous data movement without CPU intervention.

During data transfers, the CPU initiates a DMA request, allowing data to be copied to accelerator memory in the background. Similarly, result transfers back to main memory occur without blocking execution. This enables overlapping computation with data movement, reducing idle time and improving accelerator utilization.

DMA is essential for enabling asynchronous data movement, which allows transfers to overlap with computation. Instead of waiting for transfers to complete before execution begins, AI workloads can stream data into the accelerator while earlier computations are still in progress, reducing idle time and improving accelerator utilization.

Unified Memory. While PCIe, NVLink, and DMA optimize explicit memory transfers, some AI workloads require a more flexible memory model that eliminates the need for manual data copying. Unified Memory provides an abstraction that allows both the host and accelerator to access a single, shared memory space, automatically handling data movement when needed.

With Unified Memory, data does not need to be explicitly copied between CPU and GPU memory before execution. Instead, when a computation requires a memory region that is currently located in host memory, the system automatically migrates it to the accelerator, handling step (1) transparently. Similarly, when computed results are accessed by the CPU, step (4) occurs automatically, eliminating the need for manual memory management.

Although Unified Memory simplifies programming, it introduces performance trade-offs. Since memory migrations occur on demand, they can lead to unpredictable latencies, particularly if large datasets need to be transferred frequently. Additionally, since Unified Memory is implemented through page migration techniques, small memory accesses can trigger excessive data movement, further reducing efficiency.

For AI workloads that require fine-grained memory control, explicit data transfers using PCIe, NVLink, and DMA often provide better performance. However, for applications where ease of development is more important than absolute speed, Unified Memory offers a convenient alternative.

11.4.3.3 Data Transfer Overheads

Host-accelerator data movement introduces overheads that impact AI workload execution. Unlike on-chip memory accesses, which occur at nanosecond latencies, host-accelerator transfers traverse system interconnects, adding latency, bandwidth constraints, and synchronization delays.

Interconnect latency affects transfer speed, with PCIe, the standard host-accelerator link, incurring significant overhead due to packet-based transactions and memory-mapped I/O. This makes frequent small transfers inefficient. Faster alternatives like NVLink reduce latency and improve bandwidth but are limited to specific hardware ecosystems.

Synchronization delays further contribute to inefficiencies. Synchronous transfers block execution until data movement completes, ensuring data consistency but introducing idle time. Asynchronous transfers allow computation and data movement to overlap, reducing stalls but requiring careful coordination to avoid execution mismatches.

These factors, including interconnect latency, bandwidth limitations, and synchronization overheads, determine AI workload efficiency. While optimization techniques mitigate these limitations, understanding these fundamental transfer mechanics is essential for improving performance.

11.4.4 Model Memory Pressure

Machine learning models impose varying memory access patterns that significantly influence accelerator performance. The way data is transferred between the host and accelerator, how frequently memory is accessed, and the efficiency of caching mechanisms all determine overall execution efficiency. While multilayer perceptrons (MLPs), convolutional neural networks (CNNs), and transformer networks each require large parameter sets, their distinct memory demands necessitate tailored optimization strategies for accelerators. Understanding these differences provides insight into why different hardware architectures exhibit varying levels of efficiency across workloads.

11.4.4.1 Multilayer Perceptrons

MLPs, also referred to as fully connected networks, are among the simplest neural architectures. Each layer consists of a dense matrix multiplication, requiring every neuron to interact with all neurons in the preceding layer. This results in high memory bandwidth demands, particularly for weights, as every input activation contributes to a large set of computations.

From a memory perspective, MLPs rely on large, dense weight matrices that frequently exceed on-chip memory capacity, necessitating off-chip memory accesses. Since accelerators cannot directly access host memory at high speed, data transfers must be explicitly managed via interconnects such as PCIe or NVLink. These transfers introduce latency and consume bandwidth, affecting execution efficiency.

Despite their bandwidth-heavy nature, MLPs exhibit regular and predictable memory access patterns, making them amenable to optimizations such as prefetching and streaming memory accesses. Dedicated AI accelerators mitigate transfer overhead by staging weight matrices in fast SRAM caches and overlapping data movement with computation through direct memory access engines, reducing execution stalls. These optimizations allow accelerators to sustain high throughput even when handling large parameter sets (Y.-H. Chen, Emer, and Sze 2017).

11.4.4.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are widely used in image processing and computer vision tasks. Unlike MLPs, which require dense matrix multiplications, CNNs process input feature maps using small filter kernels that slide across the image. This localized computation structure results in high spatial data reuse, where the same input pixels contribute to multiple convolutions.

CNN accelerators benefit from on-chip memory optimizations, as convolution filters exhibit extensive reuse, allowing weights to be stored in fast local SRAM instead of frequently accessing off-chip memory. However, activation maps require careful management due to their size. Since accessing main memory over interconnects like PCIe introduces latency and bandwidth bottlenecks, CNN accelerators employ tiling techniques to divide feature maps into smaller regions that fit within on-chip buffers. This minimizes costly external memory transfers, improving overall efficiency (Y.-H. Chen, Emer, and Sze 2017).

While CNN workloads are more memory-efficient than MLPs, managing intermediate activations remains a challenge. Accelerators use hierarchical caching strategies and DMA engines to optimize memory movement, ensuring that computations are not stalled by inefficient host-accelerator data transfers. These memory optimizations help CNN accelerators maintain high throughput by reducing reliance on off-chip memory bandwidth (Y.-H. Chen, Emer, and Sze 2017).

11.4.4.3 Transformer Networks

Transformers have become the dominant architecture for natural language processing and are increasingly used in other domains such as vision and speech recognition. Unlike CNNs, which rely on local computations, transformers perform global attention mechanisms, where each token in an input sequence can interact with all other tokens. This leads to irregular and bandwidth-intensive memory access patterns, as large key-value matrices must be fetched and updated frequently.

These models are particularly challenging for accelerators due to their massive parameter sizes, which often exceed on-chip memory capacity. As a result, frequent memory transfers between host and accelerator introduce substantial latency overheads, particularly when relying on interconnects such as PCIe. Unified Memory architectures can mitigate some of these issues by dynamically handling data movement, but they introduce additional latency due to unpredictable on-demand memory migrations. Because transformers are memory-bound rather than compute-bound, accelerators optimized for them rely on high-bandwidth memory, tensor tiling, and memory partitioning to sustain performance (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, Dhariwal, et al. 2020).

Additionally, attention caching mechanisms and specialized tensor layouts reduce redundant memory fetches, improving execution efficiency. Given the bandwidth limitations of traditional interconnects, NVLink-enabled architectures offer significant advantages for large-scale transformer training, as they provide higher throughput and lower latency compared to PCIe. Furthermore, DMA-based asynchronous memory transfers enable overlapping computation with data movement, reducing execution stalls (D. Narayanan et al. 2021a).

11.4.5 ML Accelerators Implications

The diverse memory requirements of MLPs, CNNs, and Transformers highlight the need to tailor memory architectures to specific workloads. Table 11.9 compares the memory access patterns across different models.

Table 11.9: Memory access characteristics across different ML models.

Model Type	Weight Size	Activation Reuse	Memory Access Pattern	Primary Bottleneck
MLP (Dense)	Large, dense	Low	Regular, sequential (streamed)	Bandwidth (off-chip)
CNN	Small, reused	High	Spatial locality	Feature map movement
Transformer	Massive, sparse	Low	Irregular, high-bandwidth	Memory capacity + Interconnect

Each model type presents unique challenges that directly impact accelerator design. MLPs benefit from fast streaming access to dense weight matrices, making memory bandwidth a critical factor in performance, especially when transferring large weights from host memory to accelerator memory. CNNs, with their high activation reuse and structured memory access patterns, can leverage on-chip caching and tiling strategies to minimize off-chip memory transfers. Transformers, however, impose significant demands on both bandwidth and capacity, as attention mechanisms require frequent access to large key-value matrices, leading to high interconnect traffic and increased memory pressure.

To address these challenges, modern AI accelerators incorporate multi-tier memory hierarchies that balance speed, capacity, and energy efficiency. On-chip SRAM caches and scratchpad memories store frequently accessed data, while high-bandwidth external memory provides scalability for large models. Efficient interconnects, such as NVLink, help alleviate host-accelerator transfer bottlenecks, particularly in transformer workloads where memory movement constraints can dominate execution time.

As ML workloads continue to grow in complexity, memory efficiency is becoming as critical as raw compute power. Efficient data movement strategies, asynchronous memory transfers (DMA), and unified memory architectures play a fundamental role in sustaining high performance. The following section explores the design of memory hierarchies in AI accelerators, detailing how different levels of memory interact to optimize execution efficiency.

11.5 Neural Networks Mapping

Efficient execution of machine learning models on specialized AI acceleration hardware requires a structured approach to computation, ensuring that available resources are fully utilized while minimizing performance bottlenecks. Unlike general-purpose processors, which rely on dynamic task scheduling, AI accelerators operate under a structured execution model that maximizes throughput by carefully assigning computations to processing elements. This process, known as mapping, dictates how computations are distributed across hardware resources, influencing execution speed, memory access patterns, and overall efficiency.

> Definition of Mapping in AI Acceleration
>
> **Mapping in AI Acceleration** refers to the *assignment of machine learning computations* to *hardware processing units* to optimize *execution efficiency*. This process involves *spatial allocation*, which distributes computations across *processing elements*; *temporal scheduling*, which sequences operations to maintain *balanced workloads*; and *memory-aware execution*, which strategically places *data* to minimize *access latency*. Effective mapping ensures *high resource utilization, reduced memory stalls, and energy-efficient execution*, making it a critical factor in *AI acceleration*.

Mapping machine learning models onto AI accelerators presents several challenges due to hardware constraints and the diversity of model architectures. Given the hierarchical memory system of modern accelerators, mapping strategies must carefully manage when and where data is accessed to minimize latency and power overhead while ensuring that compute units remain actively engaged. Poor mapping decisions can lead to underutilized compute resources, excessive data movement, and increased execution time, ultimately reducing overall efficiency.

Mapping encompasses three interrelated aspects that form the foundation of effective AI accelerator design.

- **Computation Placement**: Systematically assigns operations (e.g., matrix multiplications, convolutions) to processing elements to maximize parallelism and reduce idle time.
- **Memory Allocation**: Carefully determines where model parameters, activations, and intermediate results reside within the memory hierarchy to optimize access efficiency.
- **Dataflow and Execution Scheduling**: Structures the movement of data between compute units to reduce bandwidth bottlenecks and ensure smooth, continuous execution.

Effective mapping strategies minimize off-chip memory accesses, maximize compute utilization, and efficiently manage data movement across different levels of the memory hierarchy. The following sections explore the key mapping choices that influence execution efficiency and lay the groundwork for optimization strategies that refine these decisions.

11.5.1 Computation Placement

Modern AI accelerators are designed to execute machine learning models with massive parallelism, leveraging thousands to millions of processing elements to perform computations simultaneously. However, simply having a large number of compute units is not enough—how computations are assigned to these units determines overall efficiency.

Without careful placement, some processing elements may sit idle while others are overloaded, leading to wasted resources, increased memory traffic, and reduced performance. Computation placement is the process of strategically mapping operations onto available hardware resources to sustain high throughput, minimize stalls, and optimize execution efficiency.

11.5.1.1 Computation Placement Definition

AI accelerators contain thousands to millions of processing elements, making computation placement a large-scale problem. Modern GPUs, such as the NVIDIA H100, feature over 16,000 CUDA cores and more than 500 specialized tensor cores, each designed to accelerate matrix operations (Jouppi, Young, et al. 2017c). TPUs utilize systolic arrays composed of thousands of interconnected multiply-accumulate (MAC) units, while wafer-scale processors like Cerebras' CS-2 push parallelism even further, integrating over 850,000 cores on a single chip (Systems 2021b). In these architectures, even minor inefficiencies in computation placement can lead to significant performance losses, as idle cores or excessive memory movement compound across the system.

Computation placement ensures that all processing elements contribute effectively to execution. This means that workloads must be distributed in a way that avoids imbalanced execution, where some processing elements sit idle while others remain overloaded. Similarly, placement must minimize unnecessary data movement, as excessive memory transfers introduce latency and power overheads that degrade system performance.

Neural network computations vary significantly based on the model architecture, influencing how placement strategies are applied. For example, in a convolutional neural network (CNN), placement focuses on dividing image regions across processing elements to maximize parallelism. A 256×256 image processed through thousands of GPU cores might be broken into small tiles, each mapped to a different processing unit to execute convolutional operations simultaneously. In contrast, a transformer-based model requires placement strategies that accommodate self-attention mechanisms, where each token in a sequence interacts with all others, leading to irregular and memory-intensive computation patterns. Meanwhile, Graph Neural Networks (GNNs) introduce additional complexity, as computations depend on sparse and dynamic graph structures that require adaptive workload distribution (Zheng et al. 2020).

Because computation placement directly impacts resource utilization, execution speed, and power efficiency, it is one of the most critical factors in AI acceleration. A well-placed computation can reduce latency by orders of magnitude, while a poorly placed one can render thousands of processing units underutilized. The next section explores why efficient computation placement is essential and the consequences of suboptimal mapping strategies.

11.5.1.2 Computation Placement Importance

While computation placement is a hardware-driven process, its importance is fundamentally shaped by the structure of neural network workloads. Different types of machine learning models exhibit distinct computation patterns, which directly influence how efficiently they can be mapped onto accelerators. Without careful placement, workloads can become unbalanced, memory access patterns can become inefficient, and the overall performance of the system can degrade significantly.

For models with structured computation patterns, such as CNNs, computation placement is relatively straightforward. CNNs process images using filters that are applied to small, localized regions, meaning their computations can be evenly distributed across processing elements. Because these operations are highly parallelizable, CNNs benefit from spatial partitioning, where the input is divided into tiles that are processed independently. This structured execution makes CNNs well-suited for accelerators that favor regular dataflows, minimizing the complexity of placement decisions.

However, for models with irregular computation patterns, such as transformers and GNNs, computation placement becomes significantly more challenging. Transformers, which rely on self-attention mechanisms, require each token in a sequence to interact with all others, resulting in non-uniform computation demands. Unlike CNNs, where each processing element performs a similar amount of work, transformers introduce workload imbalance, where certain operations, including the computation of attention scores, require far more computation than others. Without careful placement, this imbalance can lead to stalls, where some processing elements remain idle while others struggle to keep up.

The challenge is even greater in graph neural networks (GNNs), where computation depends on sparse and dynamically changing graph structures. Unlike CNNs, which operate on dense and regularly structured data, GNNs must process nodes and edges with highly variable degrees of connectivity. Some regions of a graph may require significantly more computation than others, making workload balancing across processing elements difficult (Zheng et al. 2020). If computations are not placed strategically, some compute units will sit idle while others remain overloaded, leading to underutilization and inefficiencies in execution.

Poor computation placement adversely affects AI execution by creating workload imbalance, inducing excessive data movement, and causing execution stalls and bottlenecks. Specifically, an uneven distribution of computations can lead to idle processing elements, thereby preventing full hardware utilization and diminishing throughput. In addition, inefficient execution assignment increases memory traffic by necessitating frequent data transfers between memory hierarchies, which in turn introduces latency and raises power consumption. Finally, such misallocation can cause operations to wait on data dependencies, resulting in pipeline inefficiencies that ultimately lower overall system performance.

Ultimately, computation placement is not just about assigning operations to processing elements—it is about ensuring that models execute efficiently given their unique computational structure. A well-placed workload reduces execution time, memory overhead, and power consumption, while a poorly placed one can lead to stalled execution pipelines and inefficient resource utilization. The next section explores the key considerations that must be addressed to ensure that computation placement is both efficient and adaptable to different model architectures.

11.5.1.3 Effective Computation Placement

Computation placement is a balancing act between hardware constraints and workload characteristics. To achieve high efficiency, placement strategies must account for parallelism, memory access, and workload variability while ensuring that processing elements remain fully utilized. Poor placement leads to imbalanced execution, increased data movement, and performance degradation, making it essential to consider key factors when designing placement strategies.

As summarized in Table 11.10, computation placement faces several critical challenges that impact execution efficiency. Effective mapping strategies must address these challenges by balancing workload distribution, minimizing data movement, and optimizing communication across processing elements.

Table 11.10: Primary challenges in computation placement and key considerations for effective mapping strategies.

Challenge	Impact on Execution	Key Considerations for Placement
Workload Imbalance	Some processing elements finish early while others remain overloaded, leading to idle compute resources.	Distribute operations evenly to prevent stalls and ensure full utilization of PEs.

Challenge	Impact on Execution	Key Considerations for Placement
Irregular Computation Patterns	Models like transformers and GNNs introduce non-uniform computation demands, making static placement difficult.	Use adaptive placement strategies that adjust execution based on workload characteristics.
Excessive Data Movement	Frequent memory transfers introduce latency and increase power consumption.	Keep frequently used data close to the compute units and minimize off-chip memory accesses.
Limited Interconnect Bandwidth	Poorly placed operations can create congestion, slowing data movement between PEs.	Optimize spatial and temporal placement to reduce communication overhead.
Model-Specific Execution Needs	CNNs, transformers, and GNNs require different execution patterns, making a single placement strategy ineffective.	Tailor placement strategies to match the computational structure of each model type.

Each of these challenges highlights a core trade-off in computation placement: maximizing parallelism while minimizing memory overhead. For CNNs, placement strategies prioritize structured tiling to maintain efficient data reuse. For transformers, placement must ensure balanced execution across attention layers. For GNNs, placement must dynamically adjust to sparse computation patterns.

Beyond model-specific needs, effective computation placement must also be scalable. As models grow in size and complexity, placement strategies must adapt dynamically rather than relying on static execution patterns. Future AI accelerators increasingly integrate runtime-aware scheduling mechanisms, where placement is optimized based on real-time workload behavior rather than predetermined execution plans.

Ultimately, effective computation placement requires a holistic approach that balances hardware capabilities with model characteristics. The next section explores how computation placement interacts with memory allocation and data movement, ensuring that AI accelerators operate at peak efficiency.

11.5.2 Memory Allocation

Efficient memory allocation is a key requirement for high-performance AI acceleration. As AI models grow in complexity, accelerators must manage vast amounts of data movement—loading model parameters, storing intermediate activations, and handling gradient computations. The way this data is allocated across the memory hierarchy directly affects execution efficiency, power consumption, and overall system throughput.

11.5.2.1 Memory Allocation Definition

While computation placement determines where operations are executed, memory allocation defines where data is stored and how it is accessed throughout execution. As discussed earlier, all AI accelerators rely on hierarchical memory systems, ranging from on-chip caches and scratchpads to HBM and DRAM. Poor memory allocation can lead to excessive off-chip memory accesses, increasing bandwidth contention and slowing down execution. Since AI accelerators operate at teraflop and petaflop scales, inefficient memory access patterns can result in substantial performance bottlenecks.

The primary goal of memory allocation is to minimize latency and reduce power consumption by keeping frequently accessed data as close as possible to the processing elements. Different hardware architectures implement memory hierarchies tailored for AI workloads. GPUs rely on a mix of global memory, shared memory, and registers, requiring careful tiling strategies to optimize locality. TPUs use on-chip SRAM scratchpads, where activations and weights must be efficiently preloaded to sustain systolic array execution. Wafer-scale processors, with their hundreds of thousands of cores, demand sophisticated memory partitioning strategies to avoid excessive interconnect traffic. In all cases, the effectiveness of memory allocation determines the overall throughput, power efficiency, and scalability of AI execution.

11.5.2.2 Memory Allocation Importance

Memory allocation is important in AI acceleration because how and where data is stored directly impacts execution efficiency. Unlike general-purpose computing, where memory management is abstracted by caches and dynamic allocation, AI accelerators require explicit data placement strategies to sustain high throughput and avoid unnecessary stalls. When memory is not allocated efficiently, AI workloads suffer from latency overhead, excessive power consumption, and bottlenecks that limit computational performance.

Neural network architectures have varying memory demands, which influence the importance of proper allocation. CNNs rely on structured and localized data access patterns, meaning that inefficient memory allocation can lead to redundant data loads and cache inefficiencies. In contrast, transformer models require frequent access to large model parameters and intermediate activations, making them highly sensitive to memory bandwidth constraints. GNNs introduce even greater challenges, as their irregular and sparse data structures result in unpredictable memory access patterns that can lead to inefficient use of memory resources. Poor memory allocation has three major consequences for AI execution:

1. **Increased Memory Latency**: When frequently accessed data is not stored in the right location, accelerators must retrieve it from higher-latency memory, slowing down execution.
2. **Higher Power Consumption**: Off-chip memory accesses consume significantly more energy than on-chip storage, leading to inefficiencies at scale.
3. **Reduced Computational Throughput**: If data is not available when needed, processing elements remain idle, reducing the overall performance of the system.

As AI models continue to grow in size and complexity, the importance of scalable and efficient memory allocation increases. Memory limitations can dictate how large of a model can be deployed on a given accelerator, affecting feasibility and performance. The next section explores the key considerations that impact memory allocation strategies and the constraints that must be addressed to optimize execution efficiency.

11.5.2.3 Effective Memory Allocation

Inefficient allocation leads to frequent stalls, excessive memory traffic, and power inefficiencies, all of which degrade overall performance. As summarized in Table 11.11, memory allocation in AI accelerators must address several key challenges that influence execution efficiency. Effective allocation strategies mitigate high latency, bandwidth limitations, and irregular access patterns by carefully managing data placement and movement. Ensuring that frequently accessed data is stored in faster memory locations while minimizing unnecessary transfers is essential for maintaining performance and energy efficiency.

Each of these challenges requires careful memory management to balance execution efficiency with hardware constraints. While structured models may benefit from well-defined memory layouts that facilitate predictable access, others, like transformer-based and graph-based models, require more adaptive allocation strategies to handle variable and complex memory demands.

Table 11.11: Key challenges in memory allocation and considerations for efficient execution.

Challenge	Impact on Execution	Key Considerations for Allocation
High Memory Latency	Slow data access delays execution and reduces throughput.	Prioritize placing frequently accessed data in faster memory locations.
Limited On-Chip Storage	Small local memory constrains the amount of data available near compute units.	Allocate storage efficiently to maximize data availability without exceeding hardware limits.
High Off-Chip Bandwidth Demand	Frequent access to external memory increases delays and power consumption.	Reduce unnecessary memory transfers by carefully managing when and how data is moved.
Irregular Memory Access Patterns	Some models require accessing data unpredictably, leading to inefficient memory usage.	Organize memory layout to align with access patterns and minimize unnecessary data movement.
Model-Specific Memory Needs	Different models require different allocation strategies to optimize performance.	Tailor allocation decisions based on the structure and execution characteristics of the workload.

Beyond workload-specific considerations, memory allocation must also be scalable. As model sizes continue to grow, accelerators must dynamically manage memory resources rather than relying on static allocation schemes. Ensuring that frequently used data is accessible when needed without overwhelming memory capacity is essential for maintaining high efficiency.

In summary, mapping neural network computations to specialized hardware is a foundational step in AI acceleration, directly influencing performance, memory efficiency, and energy consumption. However, selecting an effective mapping strategy is not a trivial task—hardware constraints, workload variability, and execution dependencies create a vast and complex design space.

While the principles of computation placement, memory allocation, and data movement provide a structured foundation, optimizing these decisions requires advanced techniques to navigate the trade-offs involved. The next section explores optimization strategies that refine mapping decisions, focusing on techniques that efficiently search the design space to maximize execution efficiency while balancing hardware constraints.

11.5.3 Combinatorial Complexity

The efficient execution of machine learning models on AI accelerators requires careful consideration of placement and allocation. Placement involves spatial assignment of computations and data, while allocation covers temporal distribution of resources. These decisions are interdependent, and each introduces trade-offs that impact performance, energy efficiency, and scalability. Table 11.12 outlines the fundamental trade-offs between computation placement and resource allocation in AI accelerators. Placement decisions influence parallelism, memory access patterns, and communication overhead, while allocation strategies determine how resources are distributed over time to balance execution efficiency. The interplay between these factors shapes overall performance, requiring a careful balance to avoid bottlenecks such as excessive synchronization, memory congestion, or underutilized compute resources. Optimizing these trade-offs is essential for ensuring that AI accelerators operate at peak efficiency.

Table 11.12: Trade-offs between computation placement and resource allocation in AI accelerators.

Dimension	Placement Considerations	Allocation Considerations
Computational Granularity	Fine-grained placement enables greater parallelism but increases synchronization overhead.	Coarse-grained allocation reduces synchronization overhead but may limit flexibility.
Spatial vs. Temporal Mapping	Spatial placement enhances parallel execution but can lead to resource contention and memory congestion.	Temporal allocation balances resource sharing but may reduce overall throughput.
Memory and Data Locality	Placing data closer to compute units minimizes latency but may reduce overall memory availability.	Allocating data across multiple memory levels increases capacity but introduces higher access costs.
Communication and Synchronization	Co-locating compute units reduces communication latency but may introduce contention.	Allocating synchronization mechanisms mitigates stalls but can introduce additional overhead.
Dataflow and Execution Ordering	Static placement simplifies execution but limits adaptability to workload variations.	Dynamic allocation improves adaptability but adds scheduling complexity.

Each of these dimensions requires balancing trade-offs between placement and allocation. For instance, spatially distributing computations across multiple processing elements can increase throughput; however, if data allocation is not optimized, memory bandwidth limitations may introduce bottlenecks. Likewise, allocating resources for fine-grained computations may enhance flexibility but, without appropriate placement strategies, may lead to excessive synchronization overhead.

Because AI accelerator architectures impose constraints on both where computations execute and how resources are assigned over time, selecting an effective mapping strategy necessitates a coordinated approach to placement and allocation. Understanding how these trade-offs influence execution efficiency is essential for optimizing performance on AI accelerators.

11.5.3.1 Configuration Space Mapping

The efficiency of AI accelerators is determined not only by their computational capabilities but also by how neural network computations are mapped to hardware resources. Mapping defines how computations are assigned to processing elements, how data is placed and moved through the memory hierarchy, and how execution is scheduled. The choices made in this process significantly impact performance, influencing compute utilization, memory bandwidth efficiency, and energy consumption.

Mapping machine learning models to hardware presents a large and complex design space. Unlike traditional computational workloads, model execution involves multiple interacting factors, including computation, data movement, parallelism, and scheduling, each introducing constraints and trade-offs. The hierarchical memory structure of accelerators, as discussed in the Memory Systems section, further complicates this process by imposing limits on bandwidth, latency, and data reuse. As a result, effective mapping strategies must carefully balance competing objectives to maximize efficiency.

At the heart of this design space lie three interconnected aspects: data placement, computation scheduling, and data movement timing. Data placement refers to the allocation of data across various memory hierarchies, such as on-chip buffers, caches, and off-chip DRAM, and its effective management is critical because it influences both latency and energy consumption. Inefficient placement often results in frequent, costly memory accesses, whereas strategic placement ensures that data used regularly remains in fast-access storage. Computation scheduling governs the order in which operations execute, impacting compute efficiency and memory access patterns; for instance, some execution orders may optimize parallelism while introducing synchronization overheads, and others may improve data locality at the expense of throughput. Meanwhile, timing in data movement is equally essential, as transferring data between memory levels incurs significant latency and energy costs. Efficient mapping strategies thus focus on minimizing unnecessary transfers by reusing data and overlapping communication with computation to enhance overall performance.

These factors define a vast combinatorial design space, where small variations in mapping decisions can lead to large differences in performance and energy efficiency. A poor mapping strategy can result in underutilized compute resources, excessive data movement, or imbalanced workloads, creating bottlenecks that degrade overall efficiency. Conversely, a well-designed mapping maximizes both throughput and resource utilization, making efficient use of available hardware.

Because of the interconnected nature of mapping decisions, there is no single optimal solution—different workloads and hardware architectures demand different approaches. The next sections examine the structure of this design space and how different mapping choices shape the execution of machine learning workloads.

Mapping machine learning computations onto specialized hardware requires balancing multiple constraints, including compute efficiency, memory bandwidth, and execution scheduling. The challenge arises from the vast number of possible ways to assign computations to processing elements, order execution, and manage data movement. Each decision contributes to a high-dimensional search space, where even minor variations in mapping choices can significantly impact performance.

Unlike traditional workloads with predictable execution patterns, machine learning models introduce diverse computational structures that require flexible mappings adapted to data reuse, parallelization opportunities, and memory constraints. The search space grows combinatorially, making exhaustive search infeasible. To understand this complexity, we analyze three key sources of variation:

11.5.3.2 Computation and Execution Ordering

Machine learning workloads are often structured as nested loops, iterating over various dimensions of computation. For instance, a matrix multiplication kernel may loop over batch size (N), input features (C), and output features (K). The order in which these loops execute has a profound effect on data locality, reuse patterns, and computational efficiency.

The number of ways to arrange d loops follows a factorial growth pattern:

$$\mathcal{O} = d!$$

which scales rapidly. A typical convolutional layer may involve up to seven loop dimensions, leading to:

$$7! = 5{,}040 \text{ possible execution orders.}$$

Furthermore, when considering multiple memory levels, the search space expands as:

$$(d!)^l$$

where l is the number of memory hierarchy levels. This rapid expansion highlights why execution order optimization is crucial—poor loop ordering can lead to excessive memory traffic, while an optimized order improves cache utilization (Sze et al. 2017a).

11.5.3.3 Processing Elements Parallelization

Modern AI accelerators leverage thousands of processing elements to maximize parallelism, but determining which computations should be parallelized is non-trivial. Excessive parallelization can introduce synchronization overheads and increased bandwidth demands, while insufficient parallelization leads to underutilized hardware.

The number of ways to distribute computations among parallel units follows the binomial coefficient:

$$\mathcal{P} = \frac{d!}{(d-k)!}$$

where d is the number of loops, and k is the number selected for parallel execution. For a six-loop computation where three loops are chosen for parallel execution, the number of valid configurations is:

$$\frac{6!}{(6-3)!} = 120.$$

Even for a single layer, there can be hundreds of valid parallelization strategies, each affecting data synchronization, memory contention, and overall compute efficiency. Expanding this across multiple layers and model architectures further magnifies the complexity.

11.5.3.4 Memory Placement and Data Movement

The hierarchical memory structure of AI accelerators introduces additional constraints, as data must be efficiently placed across registers, caches, shared memory, and off-chip DRAM. Data placement impacts latency, bandwidth consumption, and energy efficiency—frequent access to slow memory creates bottlenecks, while optimized placement reduces costly memory transfers.

The number of ways to allocate data across memory levels follows an exponential growth function:

$$\mathcal{M} = n^{d \times l}$$

where:

- n = number of placement choices per level,
- d = number of computational dimensions,
- l = number of memory hierarchy levels.

For a model with:

- $d = 5$ computational dimensions,
- $l = 3$ memory levels,
- $n = 4$ possible placement choices per level,

the number of possible memory allocations is:

$$4^{5 \times 3} = 4^{15} = 1,073,741,824.$$

This highlights how even a single layer may have over a billion possible memory configurations, making manual optimization impractical.

11.5.3.5 Mapping Search Space

By combining the complexity from computation ordering, parallelization, and memory placement, the total mapping search space can be approximated as:

$$\mathcal{S} = \left(n^d \times d! \times \frac{d!}{(d-k)!} \right)^l$$

where:

- n^d represents memory placement choices,
- $d!$ accounts for computation ordering choices,
- $\frac{d!}{(d-k)!}$ captures parallelization possibilities,
- l is the number of memory hierarchy levels.

This equation illustrates the exponential growth of the search space, making brute-force search infeasible for all but the simplest cases.

11.6 Optimization Strategies

Efficiently mapping machine learning computations onto hardware is a complex challenge due to the vast number of possible configurations. As models grow in complexity, the number of potential mappings increases exponentially. Even for a single layer, there are thousands of ways to order computation loops, hundreds of parallelization strategies, and an exponentially growing number of memory placement choices. This combinatorial explosion makes exhaustive search impractical.

To overcome this challenge, AI accelerators rely on structured mapping strategies that systematically balance computational efficiency, data locality, and parallel execution. Rather than evaluating every possible configuration, these approaches use a combination of heuristic, analytical, and machine learning-based techniques to find high-performance mappings efficiently.

The key to effective mapping lies in understanding and applying a set of core techniques that optimize data movement, memory access, and computation. These building blocks of mapping strategies provide a structured foundation for efficient execution, which we explore in the next section.

11.6.1 Mapping Strategies Building Blocks

To navigate the complexity of mapping decisions, a set of foundational techniques is leveraged that optimizes execution across data movement, memory access, and computation efficiency. These techniques provide the necessary structure for mapping strategies that maximize hardware performance while minimizing bottlenecks.

Key techniques include data movement strategies, which determine where data is staged during computation in order to reduce redundant transfers, such as in weight stationary, output stationary, and input stationary approaches. Memory-aware tensor layouts also play an important role by influencing memory access patterns and cache efficiency through the organization of data in formats such as row-major or channel-major.

Other strategies involve kernel fusion, a method that minimizes redundant memory writes by combining multiple operations into a single computational step. Tiling is employed as a technique that partitions large computations into smaller, memory-friendly blocks to improve cache efficiency and reduce memory bandwidth requirements. Finally, balancing computation and communication is essential for managing the trade-offs between parallel execution and memory access to achieve high throughput.

Each of these building blocks plays a crucial role in structuring high-performance execution, forming the basis for both heuristic and model-driven optimization techniques. In the next section, we explore how these strategies are adapted to different types of AI models.

11.6.1.1 Data Movement Patterns

While computational mapping determines where and when operations occur, its success depends heavily on how efficiently data is accessed and transferred across the memory hierarchy. Unlike traditional computing workloads, which often exhibit structured and predictable memory access patterns, machine learning workloads present irregular access behaviors due to frequent retrieval of weights, activations, and intermediate values.

Even when computational units are mapped efficiently, poor data movement strategies can severely degrade performance, leading to frequent memory stalls and underutilized hardware resources. If data cannot be supplied to processing elements at the required rate, computational units remain idle, increasing latency, memory traffic, and energy consumption (Y.-H. Chen et al. 2016).

To illustrate the impact of data movement inefficiencies, consider a typical matrix multiplication operation shown in Listing 11.18, which forms the backbone of many machine learning models.

Listing 11.18: Matrix multiplication illustrating data movement bottlenecks

```
## Matrix multiplication where:
## weights: [512 x 256] - model parameters
## input:   [256 x 32]  - batch of activations
## Z:       [512 x 32]  - output activations

## Computing each output element Z[i,j]:
for i in range(512):
    for j in range(32):
        for k in range(256):
            Z[i,j] += weights[i,k] * input[k,j]
```

This computation reveals several critical dataflow challenges.

The first challenge is the number of memory accesses required. For each output $Z[i,j]$, the computation must fetch an entire row of weights from the weight matrix and a full column of activations from the input matrix. Since the weight matrix contains 512 rows and the input matrix contains 32 columns, this results in repeated memory accesses that place a significant burden on memory bandwidth.

The second challenge comes from weight reuse. The same weights are applied to multiple inputs, meaning that an ideal mapping strategy should maximize weight locality to avoid redundant memory fetches. Without proper reuse, the accelerator would waste bandwidth loading the same weights multiple times (Tianqi et al. 2018).

The third challenge involves the accumulation of intermediate results. Since each element in $Z[i,j]$ requires contributions from 256 different weight-input pairs, partial sums must be stored and retrieved before the final value is computed. If these intermediate values are stored inefficiently, the system will require frequent memory accesses, further increasing bandwidth demands.

A natural way to mitigate these challenges is to leverage SIMD and SIMT execution models, which allow multiple values to be fetched in parallel. However, even with these optimizations, data movement remains a bottleneck. The issue is not just how quickly data is retrieved but how often it must be moved and where it is placed within the memory hierarchy (Han et al. 2016).

To address these constraints, accelerators implement dataflow strategies that determine which data remains fixed in memory and which data is streamed dynamically. These strategies aim to maximize reuse of frequently accessed data, thereby reducing the need for redundant memory fetches. The effectiveness of a given dataflow strategy depends on the specific workload—for example, deep convolutional networks benefit from keeping weights stationary, while fully connected layers may require a different approach.

Weight Stationary. The Weight Stationary strategy keeps weights fixed in local memory, while input activations and partial sums are streamed through the system. This approach is particularly beneficial in CNNs and matrix multiplications, where the same set of weights is applied across multiple inputs. By ensuring weights remain stationary, this method reduces redundant memory fetches, which helps alleviate bandwidth bottlenecks and improves energy efficiency.

A key advantage of the weight stationary approach is that it maximizes weight reuse, reducing the frequency of memory accesses to external storage. Since weight parameters are often shared across multiple computations, keeping them in local memory eliminates unnecessary data movement, lowering the overall energy cost of computation. This makes it particularly effective for architectures where weights represent the dominant memory overhead, such as systolic arrays and custom accelerators designed for machine learning.

A simplified Weight Stationary implementation for matrix multiplication is illustrated in Listing 11.19.

Listing 11.19: Weight Stationary implementation for matrix multiplication

```
## Weight Stationary Matrix Multiplication
## - Weights remain fixed in local memory
## - Input activations stream through
## - Partial sums accumulate for final output

for weight_block in weights: # Load and keep weights stationary
    load_to_local(weight_block)  # Fixed in local storage
    for input_block in inputs:   # Stream inputs dynamically
        for output_block in outputs:  # Compute results
            output_block += compute(weight_block, input_block)
            # Reuse weights across inputs
```

In weight stationary execution, weights are loaded once into local memory and remain fixed throughout the computation, while inputs are streamed dynamically, thereby reducing redundant memory accesses. At the same time, partial sums are accumulated in an efficient manner that minimizes unnecessary data movement, ensuring that the system maintains high throughput and energy efficiency.

By keeping weights fixed in local storage, memory bandwidth requirements are significantly reduced, as weights do not need to be reloaded for each new computation. Instead, the system efficiently reuses the stored weights across multiple input activations, allowing for high throughput execution. This makes weight stationary dataflow highly effective for workloads with heavy weight reuse patterns, such as CNNs and matrix multiplications.

However, while this strategy reduces weight-related memory traffic, it introduces trade-offs in input and output movement. Since inputs must be streamed dynamically while weights remain fixed, the efficiency of this approach depends on how well input activations can be delivered to the computational units without causing stalls. Additionally, partial sums, which represent intermediate results, must be carefully accumulated to avoid excessive memory traffic. The total performance gain depends on the size of available on-chip memory, as storing larger weight matrices locally can become a constraint in models with millions or billions of parameters.

The weight stationary strategy is well-suited for workloads where weights exhibit high reuse and memory bandwidth is a limiting factor. It is commonly employed in CNNs, systolic arrays, and matrix multiplication kernels, where structured weight reuse leads to significant performance improvements. However, for models where input or output reuse is more critical, alternative dataflow strategies, such as output stationary or input stationary, may provide better trade-offs.

Output Stationary. The Output Stationary strategy keeps partial sums fixed in local memory, while weights and input activations stream through the system. This approach is particularly effective for fully connected layers, systolic arrays, and other operations where an output element accumulates contributions from multiple weight-input pairs. By keeping partial sums stationary, this method reduces redundant memory writes, minimizing bandwidth consumption and improving energy efficiency (Y.-H. Chen et al. 2016).

A key advantage of the output stationary approach is that it optimizes accumulation efficiency, ensuring that each output element is computed as efficiently as possible before being written to memory. Unlike Weight Stationary, which prioritizes weight reuse, Output Stationary execution is designed to minimize memory bandwidth overhead caused by frequent writes of intermediate results. This makes it well-suited for workloads where accumulation dominates the computational pattern, such as fully connected layers and matrix multiplications in transformer-based models.

Listing 11.20 shows a simplified Output Stationary implementation for matrix multiplication.

Listing 11.20: Output Stationary implementation for matrix multiplication

```
## - Partial sums remain in local memory
## - Weights and input activations stream through dynamically
## - Final outputs are written only once

for output_block in outputs:  # Keep partial sums stationary
    accumulator = 0           # Initialize accumulation buffer
    for weight_block, input_block in zip(weights, inputs):
        accumulator += compute(weight_block, input_block)
        # Accumulate partial sums
    store_output(accumulator)  # Single write to memory
```

This implementation follows the core principles of output stationary execution: - Partial sums are kept in local memory throughout the computation. - Weights and inputs are streamed dynamically, ensuring that intermediate results remain locally accessible. - Final outputs are written back to memory only once, reducing unnecessary memory traffic.

By accumulating partial sums locally, this approach eliminates excessive memory writes, improving overall system efficiency. In architectures such as systolic arrays, where computation progresses through a grid of processing elements, keeping partial sums stationary aligns naturally with structured accumulation workflows, reducing synchronization overhead.

However, while Output Stationary reduces memory write traffic, it introduces trade-offs in weight and input movement. Since weights and activations must be streamed dynamically, the efficiency of this approach depends on how well data can be fed into the system without causing stalls. Additionally, parallel implementations must carefully synchronize updates to partial sums, especially in architectures where multiple processing elements contribute to the same output.

The Output Stationary strategy is most effective for workloads where accumulation is the dominant operation and minimizing intermediate memory writes is critical. It is commonly employed in fully connected layers, attention mechanisms, and systolic arrays, where structured accumulation leads to significant performance improvements. However, for models where input reuse is more critical, alternative dataflow strategies, such as Input Stationary, may provide better trade-offs.

Input Stationary. The Input Stationary strategy keeps input activations fixed in local memory, while weights and partial sums stream through the system. This approach is particularly effective for batch processing, transformer models, and sequence-based architectures, where input activations are reused across multiple computations. By ensuring that activations remain in local memory, this method reduces redundant input fetches, improving data locality and minimizing memory traffic.

A key advantage of the Input Stationary approach is that it maximizes input reuse, reducing the frequency of memory accesses for activations. Since many models, especially those in natural language processing (NLP) and recommendation systems, process the same input data across multiple computations, keeping inputs stationary eliminates unnecessary memory transfers, thereby lowering energy consumption. This strategy is particularly useful when dealing with large batch sizes, where a single batch of input activations contributes to multiple weight transformations.

A simplified Input Stationary implementation for matrix multiplication is illustrated in Listing 11.21.

This implementation follows the core principles of input stationary execution:

- **Input activations are loaded into local memory** and remain fixed during computation.
- **Weights are streamed dynamically**, ensuring efficient application across multiple inputs.

Listing 11.21: Input Stationary implementation for matrix multiplication

```
## - Input activations remain in local memory
## - Weights stream through dynamically
## - Partial sums accumulate and are written out

for input_block in inputs:   # Keep input activations stationary
    load_to_local(input_block)  # Fixed in local storage
    for weight_block in weights:   # Stream weights dynamically
        for output_block in outputs:  # Compute results
            output_block += compute(weight_block, input_block)
            # Reuse inputs across weights
```

- **Partial sums are accumulated and written out**, optimizing memory bandwidth usage.

By keeping input activations stationary, this strategy minimizes redundant memory accesses to input data, significantly reducing external memory bandwidth requirements. This is particularly beneficial in transformer architectures, where each token in an input sequence is used across multiple attention heads and layers. Additionally, in batch processing scenarios, keeping input activations in local memory improves data locality, making it well-suited for fully connected layers and matrix multiplications.

However, while Input Stationary reduces memory traffic for activations, it introduces trade-offs in weight and output movement. Since weights must be streamed dynamically while inputs remain fixed, the efficiency of this approach depends on how well weights can be delivered to the computational units without causing stalls. Additionally, partial sums must be accumulated efficiently before being written back to memory, which may require additional buffering mechanisms.

The Input Stationary strategy is most effective for workloads where input activations exhibit high reuse, and memory bandwidth for inputs is a critical constraint. It is commonly employed in transformers, recurrent networks, and batch processing workloads, where structured input reuse leads to significant performance improvements. However, for models where output accumulation is more critical, alternative dataflow strategies, such as Output Stationary, may provide better trade-offs.

11.6.1.2 Memory-Aware Tensor Layouts

Efficient execution of machine learning workloads depends not only on how data moves (dataflow strategies) but also on how data is stored and accessed in memory. Tensor layouts, which refers to the arrangement of multidimensional data in memory, can significantly impact memory access efficiency, cache performance, and computational throughput. Poorly chosen layouts can lead to excessive memory stalls, inefficient cache usage, and increased data movement costs.

In AI accelerators, tensor layout optimization is particularly important because data is frequently accessed in patterns dictated by the underlying hardware architecture. Choosing the right layout ensures that memory accesses align with hardware-friendly access patterns, minimizing overhead from costly memory transactions (N. Corporation 2021).

While developers can sometimes manually specify tensor layouts, the choice is often determined automatically by machine learning frameworks (e.g., TensorFlow, PyTorch, JAX), compilers, or AI accelerator runtimes. Low-level optimization tools such as cuDNN (for NVIDIA GPUs), XLA (for TPUs), and MLIR (for custom accelerators) may rearrange tensor layouts dynamically to optimize performance (X. He 2023a). In high-level frameworks, layout transformations are typically applied transparently, but developers working with custom kernels or low-level libraries (e.g., CUDA, Metal, or OpenCL) may have direct control over tensor format selection.

For example, in PyTorch, users can manually modify layouts using tensor.permute() or tensor.contiguous() to ensure efficient memory access (Paszke et al. 2019). In TensorFlow, layout optimizations are often applied internally by the XLA compiler, choosing between NHWC (row-major) and NCHW (channel-major) based on the target hardware (Brain 2022). Hardware-aware machine learning libraries, such as cuDNN for GPUs or OneDNN for CPUs, enforce specific memory layouts to maximize cache locality and SIMD efficiency. Ultimately, while developers may have some control over tensor layout selection, most layout decisions are driven by the compiler and runtime system, ensuring that tensors are stored in memory in a way that best suits the underlying hardware.

Row-Major Layout. Row-major layout refers to the way multi-dimensional tensors are stored in memory, where elements are arranged row by row, ensuring that all values in a given row are placed contiguously before moving to the next row. This storage format is widely used in general-purpose CPUs and some machine learning frameworks because it aligns naturally with sequential memory access patterns, making it more cache-efficient for certain types of operations (I. Corporation 2021).

To understand how row-major layout works, consider a single RGB image represented as a tensor of shape (Height, Width, Channels). If the image has a size of 3×3 pixels with 3 channels (RGB), the corresponding tensor is structured as (3, 3, 3). The values are stored in memory as follows:

$$I(0,0,0), I(0,0,1), I(0,0,2), I(0,1,0), I(0,1,1),$$
$$I(0,1,2), I(0,2,0), I(0,2,1), I(0,2,2), \ldots$$

Each row is stored contiguously, meaning all pixel values in the first row are placed sequentially in memory before moving on to the second row. This ordering is advantageous because CPUs and cache hierarchies are optimized for sequential memory access. When data is accessed in a row-wise fashion, such as when applying element-wise operations like activation functions or basic arithmetic transformations, memory fetches are efficient, and cache utilization is maximized (Sodani 2015).

The efficiency of row-major storage becomes particularly evident in CPU-based machine learning workloads, where operations such as batch normalization, matrix multiplications, and element-wise arithmetic frequently process rows of data sequentially. Since modern CPUs employ cache prefetching mechanisms, a row-major layout allows the next required data values to be preloaded into cache ahead of execution, reducing memory latency and improving overall computational throughput.

However, row-major layout can introduce inefficiencies when performing operations that require accessing data across channels rather than across rows. Consider a convolutional layer that applies a filter across multiple channels of an input image. Since channel values are interleaved in row-major storage, the convolution operation must jump across memory locations to fetch all the necessary channel values for a given pixel. These strided memory accesses can be costly on hardware architectures that rely on vectorized execution and coalesced memory access, such as GPUs and TPUs.

Despite these limitations, row-major layout remains a dominant storage format in CPU-based machine learning frameworks. TensorFlow, for instance, defaults to the NHWC (row-major) format on CPUs, ensuring that cache locality is optimized for sequential processing. However, when targeting GPUs, frameworks often rearrange data dynamically to take advantage of more efficient memory layouts, such as channel-major storage, which aligns better with parallelized computation.

Channel-Major Layout. In contrast to row-major layout, channel-major layout arranges data in memory such that all values for a given channel are stored together before moving to the next channel. This format is particularly beneficial for GPUs, TPUs, and other AI accelerators, where vectorized operations and memory coalescing significantly impact computational efficiency.

To understand how channel-major layout works, consider the same RGB image tensor of size (Height, Width, Channels) = (3, 3, 3). Instead of storing pixel values row by row, the data is structured channel-first in memory as follows:

$$\begin{gathered} I(0,0,0), I(1,0,0), I(2,0,0), I(0,1,0), I(1,1,0), I(2,1,0), \ldots, \\ I(0,0,1), I(1,0,1), I(2,0,1), \ldots, I(0,0,2), I(1,0,2), I(2,0,2), \ldots \end{gathered}$$

In this format, all red channel values for the entire image are stored first, followed by all green values, and then all blue values. This ordering allows hardware accelerators to efficiently load and process data across channels in parallel, which is crucial for convolution operations and SIMD (Single Instruction, Multiple Data) execution models (Chetlur et al. 2014).

The advantage of channel-major layout becomes clear when performing convolutions in machine learning models. Convolutional layers process images by applying a shared set of filters across all channels. When the data is stored in a channel-major format, a convolution kernel can load an entire channel efficiently, reducing the number of scattered memory fetches. This reduces memory latency, improves throughput, and enhances data locality for matrix multiplications, which are fundamental to machine learning workloads.

Because GPUs and TPUs rely on memory coalescing, a technique in which consecutive threads fetch contiguous memory addresses, channel-major layout aligns naturally with the way these processors execute parallel computations. For example, in NVIDIA GPUs, each thread in a warp (a group of threads executed simultaneously) processes different elements of the same channel, ensuring that memory accesses are efficient and reducing the likelihood of strided memory accesses, which can degrade performance.

Despite its advantages in machine learning accelerators, channel-major layout can introduce inefficiencies when running on general-purpose CPUs. Since CPUs optimize for sequential memory access, storing all values for a single channel before moving to the next disrupts cache locality for row-wise operations. This is why many machine learning frameworks (e.g., TensorFlow, PyTorch) default to row-major (NHWC) on CPUs and channel-major (NCHW) on GPUs—optimizing for the strengths of each hardware type.

Modern AI frameworks and compilers often transform tensor layouts dynamically depending on the execution environment. For instance, TensorFlow and PyTorch automatically switch between NHWC and NCHW based on whether a model is running on a CPU, GPU, or TPU, ensuring that the memory layout aligns with the most efficient execution path.

Row-Major vs Channel-Major Layouts. Both row-major (NHWC) and channel-major (NCHW) layouts serve distinct purposes in machine learning workloads, with their efficiency largely determined by the hardware architecture, memory access patterns, and computational requirements. The choice of layout directly influences cache utilization, memory bandwidth efficiency, and processing throughput. Table 11.13 summarizes the differences between row-major (NHWC) and channel-major (NCHW) layouts in terms of performance tradeoffs and hardware compatibility.

Table 11.13: Comparison of row-major (NHWC) vs. channel-major (NCHW) layouts.

Feature	Row-Major (NHWC)	Channel-Major (NCHW)
Memory Storage Order	Pixels are stored row-by-row, channel interleaved	All values for a given channel are stored together first
Best for	CPUs, element-wise operations	GPUs, TPUs, convolution operations
Cache Efficiency	High cache locality for sequential row access	Optimized for memory coalescing across channels
Convolution Performance	Requires strided memory accesses (inefficient on GPUs)	Efficient for GPU convolution kernels
Memory Fetching	Good for operations that process rows sequentially	Optimized for SIMD execution across channels
Default in Frameworks	Default on CPUs (e.g., TensorFlow NHWC)	Default on GPUs (e.g., cuDNN prefers NCHW)

The decision to use row-major (NHWC) or channel-major (NCHW) layouts is not always made manually by developers. Instead, machine learning frameworks and AI compilers often determine the optimal layout dynamically based on the target hardware and operation type. CPUs tend to favor NHWC due to cache-friendly sequential memory access, while GPUs perform better with NCHW, which reduces memory fetch overhead for machine learning computations.

In practice, modern AI compilers such as TensorFlow's XLA and PyTorch's TorchScript perform automatic layout transformations, converting tensors between NHWC and NCHW as needed to optimize performance across different processing units. This ensures that machine learning models achieve the highest possible throughput without requiring developers to manually specify tensor layouts.

11.6.1.3 Kernel Fusion

Intermediate Memory Write. Optimizing memory access is a fundamental challenge in AI acceleration. While AI models rely on high-throughput computation, their performance is often constrained by memory bandwidth and intermediate memory writes rather than pure arithmetic operations. Every time an operation produces an intermediate result that must be written to memory and later read back, execution stalls occur due to data movement overhead.

To better understand why kernel fusion is necessary, consider a simple sequence of operations in a machine learning model. Many AI workloads, particularly those involving element-wise transformations, introduce unnecessary intermediate memory writes, leading to increased memory bandwidth consumption and reduced execution efficiency (N. Corporation 2017).

Listing 11.22 illustrates a naïve execution model in which each operation is treated as a separate kernel, meaning that each intermediate result is written to memory and then read back for the next operation.

Listing 11.22: Naïve kernel-by-kernel execution

```
import torch

## Input tensor
X = torch.randn(1024, 1024).cuda()

## Step-by-step execution (naïve approach)
X1 = torch.relu(X)          # Intermediate tensor stored
                            # in memory
X2 = torch.batch_norm(X1)   # Another intermediate tensor stored
Y  = 2.0 * X2 + 1.0         # Final result
```

Each operation produces an intermediate tensor that must be written to memory and retrieved for the next operation. On large tensors, this overhead of moving data can outweigh the computational cost of the operations (Shazeer et al. 2018). Table 11.14 illustrates the memory overhead in a naïve execution model. While only the final result Y is needed, storing multiple intermediate tensors creates unnecessary memory traffic and inefficient memory usage. This data movement bottleneck significantly impacts performance, making memory optimization crucial for AI accelerators.

Table 11.14: Memory footprint of a naïve execution model with intermediate tensor storage.

Tensor	Size (MB) for 1024 × 1024 Tensor
X	4 MB
X′	4 MB
X′′	4 MB
Y	4 MB
Total Memory	16 MB

Even though only the final result Y is needed, three additional intermediate tensors consume extra memory without contributing to final output storage. This excessive memory usage limits scalability and wastes memory bandwidth, particularly in AI accelerators where minimizing data movement is critical.

Kernel Fusion for Memory Efficiency. Kernel fusion is a key optimization technique that aims to minimize intermediate memory writes, reducing the memory footprint and bandwidth consumption of machine learning workloads (Zhihao Jia, Zaharia, and Aiken 2018).

Kernel fusion involves merging multiple computation steps into a single, optimized operation, eliminating the need for storing and reloading intermediate tensors. Instead of executing each layer or element-wise operation separately, in which each step writes its output to memory before the next step begins, fusion enables direct data propagation between operations, keeping computations within high-speed registers or local memory.

A common machine learning sequence might involve applying a nonlinear activation function (e.g., ReLU), followed by batch normalization, and then scaling the values for input to the next layer. In a naïve implementation, each of these steps generates an intermediate tensor, which is written to memory, read back, and then modified again:

$$X' = \text{ReLU}(X)X'' = \text{BatchNorm}(X')Y = \alpha \cdot X'' + \beta$$

With kernel fusion, these operations are combined into a single computation step, allowing the entire transformation to occur without generating unnecessary intermediate tensors:

$$Y = \alpha \cdot \text{BatchNorm}(\text{ReLU}(X)) + \beta$$

Table 11.15 highlights the impact of operation fusion on memory efficiency. By keeping intermediate results in registers or local memory rather than writing them to main memory, fusion significantly reduces memory traffic. This optimization is especially beneficial on highly parallel architectures like GPUs and TPUs, where minimizing memory accesses translates directly into improved execution throughput. Compared to the naïve execution model, fused execution eliminates the need for storing intermediate tensors, dramatically lowering the total memory footprint and improving overall efficiency.

Table 11.15: Reduction in memory usage through operation fusion.

Execution Model	Intermediate Tensors Stored	Total Memory Usage (MB)
Naïve Execution	X′, X′′	16 MB
Fused Execution	None	4 MB

Kernel fusion reduces total memory consumption from 16 MB to 4 MB, eliminating redundant memory writes while improving execution efficiency.

Performance Benefits and Constraints. Kernel fusion brings several key advantages that enhance memory efficiency and computation throughput. By reducing memory accesses, fused kernels ensure that intermediate values stay within registers instead of being repeatedly written to and read from memory. This significantly lowers memory traffic, which is one of the primary bottlenecks in machine learning workloads. GPUs and TPUs, in particular, benefit from kernel fusion because high-bandwidth memory is a scarce resource, and reducing memory transactions leads to better utilization of compute units (X. Qi, Kantarci, and Liu 2017).

However, not all operations can be fused. Element-wise operations, such as ReLU, batch normalization, and simple arithmetic transformations, are ideal candidates for fusion since their computations depend only on single elements from the input tensor. In contrast, operations with complex data dependencies, such as matrix multiplications and convolutions, involve global data movement, making direct fusion impractical. These operations require values from multiple input elements to compute a single output, which prevents them from being executed as a single fused kernel.

Another major consideration is register pressure. Fusing multiple operations means all temporary values must be kept in registers rather than memory. While this eliminates redundant memory writes, it also increases register demand. If a fused kernel exceeds the available registers per thread, the system must spill excess values into shared memory, introducing additional latency and potentially negating the benefits of fusion. On GPUs, where thread occupancy (the number of threads that can run in parallel) is limited by available registers, excessive fusion can reduce parallelism, leading to diminishing returns.

Different AI accelerators and compilers handle fusion in distinct ways. NVIDIA GPUs, for example, favor warp-level parallelism, where element-wise fusion is straightforward. TPUs, on the other hand, prioritize systolic array execution, which is optimized for matrix-matrix operations rather than element-wise fusion (X. Qi, Kantarci, and Liu 2017). AI compilers such as XLA (TensorFlow), TorchScript (PyTorch), TensorRT (NVIDIA), and MLIR automatically detect fusion opportunities and apply heuristics to balance memory savings and execution efficiency (X. He 2023b).

Despite its advantages, fusion is not always beneficial. Some AI frameworks allow developers to disable fusion selectively, especially when debugging performance issues or making frequent model modifications. The decision to fuse operations must consider trade-offs between memory efficiency, register usage, and hardware execution constraints to ensure that fusion leads to tangible performance improvements.

11.6.1.4 Tiling for Memory Efficiency

While modern AI accelerators offer high computational throughput, their performance is often limited by memory bandwidth rather than raw processing power. If data cannot be supplied to processing units fast enough, execution stalls occur, leading to wasted cycles and inefficient hardware utilization.

Tiling is a technique used to mitigate this issue by restructuring computations into smaller, memory-friendly subproblems. Instead of processing entire matrices or tensors at once, which leads to excessive memory traffic, tiling partitions computations into smaller blocks (tiles) that fit within fast local memory (e.g., caches, shared memory, or registers) (Lam, Rothberg, and Wolf 1991). By doing so, tiling increases data reuse, minimizes memory fetches, and improves overall computational efficiency.

A classic example of inefficient memory access is matrix multiplication, which is widely used in AI models. Without tiling, the naïve approach results in repeated memory accesses for the same data, leading to unnecessary bandwidth consumption (Listing 11.23).

Listing 11.23: Naïve matrix multiplication without tiling

```
for i in range(N):
    for j in range(N):
        for k in range(N):
            C[i, j] += A[i, k] * B[k, j]  # Repeatedly fetching
                                          # A[i, k] and B[k, j]
```

Each iteration requires loading elements from matrices A and B multiple times from memory, causing excessive data movement. As the size of the matrices increases, the memory bottleneck worsens, limiting performance.

Tiling addresses this problem by ensuring that smaller portions of matrices are loaded into fast memory, reused efficiently, and only written back to main memory when necessary. This technique is especially crucial in AI accelerators, where memory accesses dominate execution time. By breaking up large matrices into smaller tiles, as illustrated in Figure 11.8, computation can be performed more efficiently on hardware by maximizing data reuse in fast memory. In the following sections, we will explore the fundamental principles of tiling, its different strategies, and the key trade-offs involved in selecting an effective tiling approach.

Tiling Fundamentals. Tiling is based on a simple but powerful principle: instead of operating on an entire data structure at once, computations are divided into smaller tiles that fit within the available fast memory. By structuring execution around these tiles, data reuse is maximized, reducing redundant memory accesses and improving overall efficiency.

Consider matrix multiplication, a key operation in machine learning workloads. The operation computes the output matrix C from two input matrices A and B:

$$C = A \times B$$

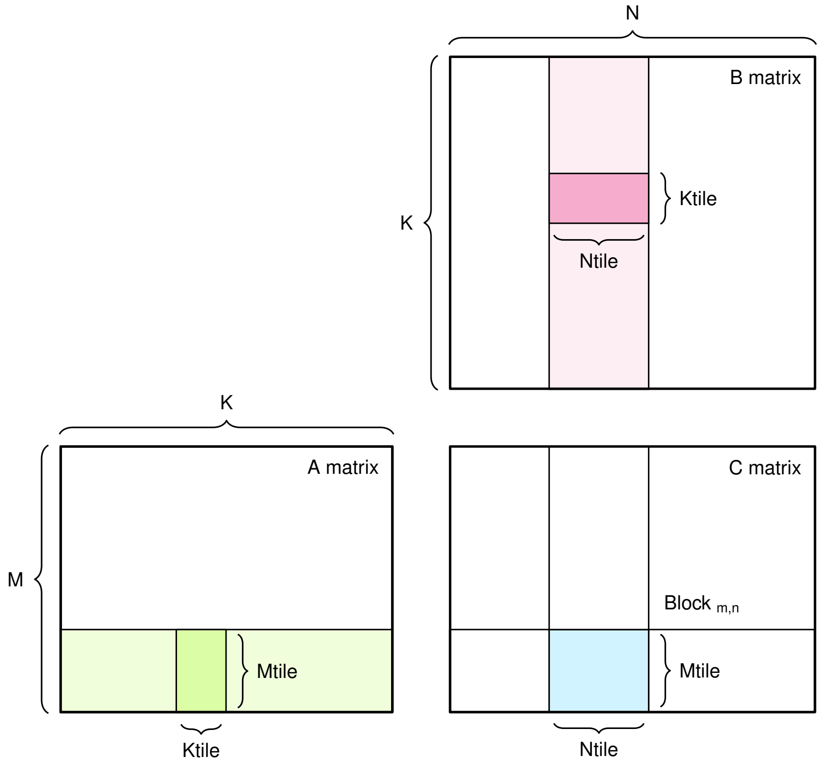


Figure 11.8: Example of how matrix multiplication can be tiled for better parallelization and memory accesses.

where each element $C[i, j]$ is computed as:

$$C[i, j] = \sum_k A[i, k] \times B[k, j]$$

A naïve implementation follows this formula directly (Listing 11.24).

Listing 11.24: Naïve matrix multiplication

```
for i in range(N):
    for j in range(N):
        for k in range(N):
            C[i, j] += A[i, k] * B[k, j]  # Repeatedly fetching
                                          # A[i, k] and B[k, j]
```

At first glance, this approach seems correct—it computes the desired result and follows the mathematical definition. However, the issue lies in how memory is accessed. Every time the innermost loop runs, it fetches an element from matrix A and matrix B from memory, performs a multiplication, and updates an element in matrix C. Because matrices are large, the processor frequently reloads the same values from memory, even though they were just used in previous computations.

This unnecessary data movement is expensive. Fetching values from main memory (DRAM) is hundreds of times slower than accessing values stored in on-chip cache or registers. If the same values must be reloaded multiple times instead of being stored in fast memory, execution slows down significantly.

Tiling Performance Improvement. Instead of computing one element at a time and constantly moving data in and out of slow memory, tiling processes submatrices (tiles) at a time, keeping frequently used values in fast memory. The idea is to divide the matrices into smaller blocks that fit within the processor's cache or shared memory, ensuring that once a block is loaded, it is reused multiple times before moving to the next one.

Listing 11.25 illustrates a tiled version of matrix multiplication, which improves memory locality by processing blocks of data.

Listing 11.25: Tiled matrix multiplication

```
TILE_SIZE = 32 # Choose a tile size based on
               # hardware constraints

for i in range(0, N, TILE_SIZE):
    for j in range(0, N, TILE_SIZE):
        for k in range(0, N, TILE_SIZE):
          # Compute the submatrix
          # C[i:i+TILE_SIZE, j:j+TILE_SIZE]
            for ii in range(i, i + TILE_SIZE):
                for jj in range(j, j + TILE_SIZE):
                    for kk in range(k, k + TILE_SIZE):
                        C[ii, jj] += A[ii, kk] * B[kk, jj]
```

This restructuring significantly improves performance for three main reasons:

1. **Better Memory Reuse**: Instead of fetching elements from A and B repeatedly from slow memory, this approach loads a small tile of data into fast memory, performs multiple computations using it, and only then moves on to the next tile. This minimizes redundant memory accesses.
2. **Reduced Memory Bandwidth Usage**: Since each tile is used multiple times before being evicted, memory traffic is reduced. Instead of repeatedly accessing DRAM, most required data is available in L1/L2 cache or shared memory, leading to faster execution.
3. **Increased Compute Efficiency**: Processors spend less time waiting for data and more time performing useful computations. In architectures like GPUs and TPUs, where thousands of parallel processing units operate simultaneously, tiling ensures that data is read and processed in a structured manner, avoiding unnecessary stalls.

This technique is particularly effective in AI accelerators, where machine learning workloads consist of large matrix multiplications and tensor transformations. Without tiling, these workloads quickly become memory-bound, meaning performance is constrained by how fast data can be retrieved rather than by the raw computational power of the processor.

Tiling Methods. While the general principle of tiling remains the same, which involves partitioning large computations into smaller subproblems to improve memory reuse, there are different ways to apply tiling based on the structure of the computation and hardware constraints. The two primary tiling strategies are spatial tiling and temporal tiling. These strategies optimize different aspects of computation and memory access, and in practice, they are often combined to achieve the best performance.

Spatial Tiling. Spatial tiling focuses on partitioning data structures into smaller blocks that fit within the fast memory of the processor. This approach ensures that each tile is fully processed before moving to the next, reducing redundant memory accesses. Spatial tiling is widely used in operations such as matrix multiplication, convolutions, and attention mechanisms in transformer models.

Spatial tiling is illustrated in Listing 11.26, where the computation proceeds over blocks of the input matrices.

Listing 11.26: Tiled matrix multiplication with spatial tiling

```
TILE_SIZE = 32  # Tile size chosen based on available
                # fast memory

for i in range(0, N, TILE_SIZE):
    for j in range(0, N, TILE_SIZE):
        for k in range(0, N, TILE_SIZE):
            # Process a submatrix (tile) at a time
            for ii in range(i, i + TILE_SIZE):
                for jj in range(j, j + TILE_SIZE):
                    for kk in range(k, k + TILE_SIZE):
                        C[ii, jj] += A[ii, kk] * B[kk, jj]
```

In this implementation, each tile of A and B is loaded into cache or shared memory before processing, ensuring that the same data does not need to be fetched repeatedly from slower memory. The tile is fully used before moving to the next block, minimizing redundant memory accesses. Since data is accessed in a structured, localized way, cache efficiency improves significantly.

Spatial tiling is particularly beneficial when dealing with large tensors that do not fit entirely in fast memory. By breaking them into smaller tiles, computations remain localized, avoiding excessive data movement between memory levels. This technique is widely used in AI accelerators where machine learning workloads involve large-scale tensor operations that require careful memory management to achieve high performance.

Temporal Tiling. While spatial tiling optimizes how data is partitioned, temporal tiling focuses on reorganizing the computation itself to improve data reuse over time. Many machine learning workloads involve operations where the same data is accessed repeatedly across multiple iterations. Without temporal tiling, this often results in redundant memory fetches, leading to inefficiencies. Temporal tiling, also known as loop blocking, restructures the computation to ensure that frequently used data stays in fast memory for as long as possible before moving on to the next computation.

A classic example where temporal tiling is beneficial is convolutional operations, where the same set of weights is applied to multiple input regions. Without loop blocking, these weights might be loaded from memory multiple times for each computation. With temporal tiling, the computation is reordered so that the weights remain in fast memory across multiple inputs, reducing unnecessary memory fetches and improving overall efficiency.

Listing 11.27 illustrates a simplified example of loop blocking in matrix multiplication.

Listing 11.27: Matrix Multiplication with Temporal Tiling (Loop Blocking)

```
for i in range(0, N, TILE_SIZE):
    for j in range(0, N, TILE_SIZE):
        for k in range(0, N, TILE_SIZE):
            # Load tile into fast memory before computation
            A_tile = A[i:i+TILE_SIZE, k:k+TILE_SIZE]
            B_tile = B[k:k+TILE_SIZE, j:j+TILE_SIZE]

            for ii in range(TILE_SIZE):
                for jj in range(TILE_SIZE):
                    for kk in range(TILE_SIZE):
                        C[i+ii, j+jj] += A_tile[ii, kk] *
                                         B_tile[kk, jj]
```

Temporal tiling improves performance by ensuring that the data loaded into fast memory is used multiple times before being evicted. In this implementation, small tiles of matrices A and B are explicitly loaded into temporary storage before performing computations, reducing memory fetch overhead. This restructuring allows the computation to process an entire tile before moving to the next, thereby reducing the number of times data must be loaded from slower memory.

This technique is particularly useful in workloads where certain values are used repeatedly, such as convolutions, recurrent neural networks (RNNs), and self-attention mechanisms in transformers. By applying loop blocking, AI accelerators can significantly reduce memory stalls and improve execution throughput.

Tiling Challenges and Trade-offs. While tiling significantly improves performance by optimizing memory reuse and reducing redundant memory accesses, it introduces several challenges and trade-offs. Selecting the right tile size is a critical decision, as it directly affects computational efficiency and memory bandwidth usage. If the tile size is too small, the benefits of tiling diminish, as memory fetches still dominate execution time. On the other hand, if the tile size is too large, it may exceed the available fast memory, causing cache thrashing and performance degradation.

Load balancing is another key concern. In architectures such as GPUs and TPUs, computations are executed in parallel across thousands of processing units. If tiles are not evenly distributed, some units may remain idle while others are overloaded, leading to suboptimal utilization of computational resources. Effective tile scheduling ensures that parallel execution remains balanced and efficient.

Data movement overhead is also an important consideration. Although tiling reduces the number of slow memory accesses, transferring tiles between different levels of memory still incurs a cost. This is especially relevant in hierarchical memory systems, where accessing data from cache is much faster than accessing it from DRAM. Efficient memory prefetching and scheduling strategies are required to minimize latency and ensure that data is available when needed.

Beyond spatial and temporal tiling, hybrid approaches combine elements of both strategies to achieve optimal performance. Hybrid tiling adapts to workload-specific constraints by dynamically adjusting tile sizes or reordering computations based on real-time execution conditions. For example, some AI accelerators use spatial tiling for matrix multiplications while employing temporal tiling for weight reuse in convolutional layers.

In addition to tiling, there are other methods for optimizing memory usage and computational efficiency. Techniques such as register blocking, double buffering, and hierarchical tiling extend the basic tiling principles to further optimize execution. AI compilers and runtime systems, such as TensorFlow XLA, TVM, and MLIR, automatically select tiling strategies based on hardware constraints, allowing for fine-tuned performance optimization without manual intervention.

Table 11.16 provides a comparative overview of spatial, temporal, and hybrid tiling approaches, highlighting their respective benefits and trade-offs.

Table 11.16: Comparative analysis of spatial, temporal, and hybrid tiling strategies.

Aspect	Spatial Tiling (Data Tiling)	Temporal Tiling (Loop Blocking)	Hybrid Tiling
Primary Goal	Reduce memory accesses by keeping data in fast memory longer	Increase data reuse across loop iterations	Adapt dynamically to workload constraints
Optimization Focus	Partitioning data structures into smaller, memory-friendly blocks	Reordering computations to maximize reuse before eviction	Balancing spatial and temporal reuse strategies
Memory Usage	Improves cache locality and reduces DRAM access	Keeps frequently used data in fast memory for multiple iterations	Minimizes data movement while ensuring high reuse
Common Use Cases	Matrix multiplications, CNNs, self-attention in transformers	Convolutions, recurrent neural networks (RNNs), iterative computations	AI accelerators with hierarchical memory, mixed workloads

Aspect	Spatial Tiling (Data Tiling)	Temporal Tiling (Loop Blocking)	Hybrid Tiling
Performance Gains	Reduced memory bandwidth requirements, better cache utilization	Lower memory fetch latency, improved data locality	Maximized efficiency across multiple hardware types
Challenges	Requires careful tile size selection, inefficient for workloads with minimal spatial reuse	Can increase register pressure, requires loop restructuring	Complexity in tuning tile size and execution order dynamically
Best When	Data is large and needs to be partitioned for efficient processing	The same data is accessed multiple times across iterations	Both data partitioning and iteration-based reuse are important

As machine learning models continue to grow in size and complexity, tiling remains a critical tool for improving hardware efficiency, ensuring that AI accelerators operate at their full potential. While manual tiling strategies can provide substantial benefits, modern compilers and hardware-aware optimization techniques further enhance performance by automatically selecting the most effective tiling strategies for a given workload.

11.6.2 Mapping Strategies Application

While the foundational mapping techniques we discussed apply broadly, their effectiveness varies based on the computational structure, data access patterns, and parallelization opportunities of different neural network architectures. Each architecture imposes distinct constraints on data movement, memory hierarchy, and computation scheduling, requiring tailored mapping strategies to optimize performance.

A structured approach to mapping is essential to address the combinatorial explosion of choices that arise when assigning computations to AI accelerators. Rather than treating each model as a separate optimization problem, we recognize that the same fundamental principles apply across different architectures—only their priority shifts based on workload characteristics. The goal is to systematically select and apply mapping strategies that maximize efficiency for different types of machine learning models.

To demonstrate these principles, we examine three representative AI workloads, each characterized by distinct computational demands. CNNs benefit from spatial data reuse, making weight-stationary execution and the application of tiling techniques especially effective. In contrast, Transformers are inherently memory-bound and rely on strategies such as efficient KV-cache management, fused attention mechanisms, and highly parallel execution to mitigate memory traffic. MLPs, which involve substantial matrix multiplication operations, demand the use of structured tiling, optimized weight layouts, and memory-aware execution to enhance overall performance.

Despite their differences, each of these models follows a common set of mapping principles, with variations in how optimizations are prioritized. The following table provides a structured mapping between different optimization strategies and their suitability for CNNs, Transformers, and MLPs. This table serves as a roadmap for selecting appropriate mapping strategies for different machine learning workloads.

Optimization Technique	CNNs	Transformers	MLPs	Rationale
Dataflow Strategy	Weight Stationary	Activation Stationary	Weight Stationary	CNNs reuse filters across spatial locations; Transformers reuse activations (KV-cache); MLPs reuse weights across batches.
Memory-Aware Tensor Layouts	NCHW (Channel-Major)	NHWC (Row-Major)	NHWC	CNNs favor channel-major for convolution efficiency; Transformers and MLPs prioritize row-major for fast memory access.
Kernel Fusion	Convolution + Activation	Fused Attention	GEMM Fusion	CNNs optimize convolution+activation fusion; Transformers fuse attention mechanisms; MLPs benefit from fused matrix multiplications.
Tiling for Memory Efficiency	Spatial Tiling	Temporal Tiling	Blocked Tiling	CNNs tile along spatial dimensions; Transformers use loop blocking to improve sequence memory efficiency; MLPs use blocked tiling for large matrix multiplications.

This table highlights that each machine learning model benefits from a different combination of optimization techniques, reinforcing the importance of tailoring execution strategies to the computational and memory characteristics of the workload.

In the following sections, we explore how these optimizations apply to each network type, explaining how CNNs, Transformers, and MLPs leverage specific mapping strategies to improve execution efficiency and hardware utilization.

11.6.2.1 Convolutional Neural Networks

CNNs are characterized by their structured spatial computations, where small filters (or kernels) are repeatedly applied across an input feature map. This structured weight reuse makes weight stationary execution the most effective strategy for CNNs. Keeping filter weights in fast memory while streaming activations ensures that weights do not need to be repeatedly fetched from slower external memory, significantly reducing memory bandwidth demands. Since each weight is applied to multiple spatial locations, weight stationary execution maximizes arithmetic intensity and minimizes redundant memory transfers.

Memory-aware tensor layouts also play a critical role in CNN execution. Convolution operations benefit from a channel-major memory format, often represented as NCHW (batch, channels, height, width). This layout aligns with the access patterns of convolutions, enabling efficient memory coalescing on accelerators such as GPUs and TPUs. By storing data in a format that optimizes cache locality, accelerators can fetch contiguous memory blocks efficiently, reducing latency and improving throughput.

Kernel fusion is another important optimization for CNNs. In a typical machine learning pipeline, convolution operations are often followed by activation functions such as ReLU and batch normalization. Instead of treating these operations as separate computational steps, fusing them into a single kernel reduces intermediate memory writes and improves execution efficiency. This optimization minimizes memory bandwidth pressure by keeping intermediate values in registers rather than writing them to memory and fetching them back in subsequent steps.

Given the size of input images and feature maps, tiling is necessary to ensure that computations fit within fast memory hierarchies. Spatial tiling, where input feature maps are processed in smaller subregions, allows for efficient utilization of on-chip memory while avoiding excessive off-chip memory transfers. This technique ensures that input activations, weights, and intermediate outputs remain within high-speed caches or shared memory as long as possible, reducing memory stalls and improving overall performance.

Together, these optimizations ensure that CNNs make efficient use of available compute resources by maximizing weight reuse, optimizing memory access patterns, reducing redundant memory writes, and structuring computation to fit within fast memory constraints.

11.6.2.2 Transformer Architectures

Unlike CNNs, which rely on structured spatial computations, Transformers process variable-length sequences and rely heavily on attention mechanisms. The primary computational bottleneck in Transformers is memory bandwidth, as attention mechanisms require frequent access to stored key-value pairs across multiple query vectors. Given this access pattern, activation stationary execution is the most effective strategy. By keeping key-value activations in fast memory and streaming query vectors dynamically, activation reuse is maximized while minimizing redundant memory fetches. This approach is critical in reducing bandwidth overhead, especially in long-sequence tasks such as natural language processing.

Memory layout optimization is equally important for Transformers. Unlike CNNs, which benefit from channel-major layouts, Transformers require efficient access to sequences of activations, making a row-major format (NHWC) the preferred choice. This layout ensures that activations are accessed contiguously in memory, reducing cache misses and improving memory coalescing for matrix multiplications.

Kernel fusion plays a key role in optimizing Transformer execution. In self-attention, multiple computational steps, such as query-key dot products, softmax normalization, and weighted summation, can be fused into a single operation. Fused attention kernels eliminate intermediate memory writes by computing attention scores and performing weighted summations within a single execution step. This optimization significantly reduces memory traffic, particularly for large batch sizes and long sequences.

Due to the nature of sequence processing, tiling must be adapted to improve memory efficiency. Instead of spatial tiling, which is effective for CNNs, Transformers benefit from temporal tiling, where computations are structured to process sequence blocks efficiently. This method ensures that activations are loaded into fast memory in manageable chunks, reducing excessive memory transfers. Temporal tiling is particularly beneficial for long-sequence models, where the memory footprint of key-value activations grows significantly. By tiling sequences into smaller segments, memory locality is improved, enabling efficient cache utilization and reducing bandwidth pressure.

These optimizations collectively address the primary bottlenecks in Transformer models by prioritizing activation reuse, structuring memory layouts for efficient batched computations, fusing attention operations to reduce intermediate memory writes, and employing tiling techniques suited to sequence-based processing.

11.6.2.3 Multi-Layer Perceptrons

MLPs primarily consist of fully connected layers, where large matrices of weights and activations are multiplied to produce output representations. Given this structure, weight stationary execution is the most effective strategy for MLPs. Similar to CNNs, MLPs benefit from keeping weights in local memory while streaming activations dynamically, as this ensures that weight matrices, which are typically reused across multiple activations in a batch, do not need to be frequently reloaded.

The preferred memory layout for MLPs aligns with that of Transformers, as matrix multiplications are more efficient when using a row-major (NHWC) format. Since activation matrices are processed in batches, this layout ensures that input activations are accessed efficiently without introducing memory fragmentation. By aligning tensor storage with compute-friendly memory access patterns, cache utilization is improved, reducing memory stalls.

Kernel fusion in MLPs is primarily applied to General Matrix Multiplication (GEMM) operations. Since dense layers are often followed by activation functions and bias additions, fusing these operations into a single computation step reduces memory traffic. GEMM fusion ensures that activations, weights, and biases are processed within a single optimized kernel, avoiding unnecessary memory writes and reloads.

To further improve memory efficiency, MLPs rely on blocked tiling strategies, where large matrix multiplications are divided into smaller sub-blocks that fit within the accelerator's shared memory. This method ensures that frequently accessed portions of matrices remain in fast memory throughout computation, reducing external memory accesses. By structuring computations in a way that balances memory utilization with efficient parallel execution, blocked tiling minimizes bandwidth limitations and maximizes throughput.

These optimizations ensure that MLPs achieve high computational efficiency by structuring execution around weight reuse, optimizing memory layouts for dense matrix operations, reducing redundant memory writes through kernel fusion, and employing blocked tiling strategies to maximize on-chip memory utilization.

11.6.3 Hybrid Mapping Strategies

While general mapping strategies provide a structured framework for optimizing machine learning models, real-world architectures often involve diverse computational requirements that cannot be effectively addressed with a single, fixed approach. Hybrid mapping strategies allow AI accelerators to dynamically apply different optimizations to specific layers or components within a model, ensuring that each computation is executed with maximum efficiency.

Machine learning models typically consist of multiple layer types, each exhibiting distinct memory access patterns, data reuse characteristics, and parallelization opportunities. By tailoring mapping strategies to these specific properties, hybrid approaches achieve higher computational efficiency, improved memory bandwidth utilization, and reduced data movement overhead compared to a uniform mapping approach (Sze et al. 2017b).

11.6.3.1 Layer-Specific Mapping

Hybrid mapping strategies are particularly beneficial in models that combine spatially localized computations, such as convolutions, with fully connected operations, such as dense layers or attention mechanisms. These operations possess distinct characteristics that require different mapping strategies for optimal performance.

In convolutional neural networks, hybrid strategies are frequently employed to optimize performance. Specifically, weight stationary execution is applied to convolutional layers, ensuring that filters remain in local memory while activations are streamed dynamically. For fully connected layers, output stationary execution is utilized to minimize redundant memory writes during matrix multiplications. Additionally, kernel fusion is integrated to combine activation functions, batch normalization, and element wise operations into a single computational step, thereby reducing intermediate memory traffic. Collectively, these approaches enhance computational efficiency and memory utilization, contributing to the overall performance of the network.

Transformers employ several strategies to enhance performance by optimizing memory usage and computational efficiency. Specifically, they use activation stationary mapping in self-attention layers to maximize the reuse of stored key-value pairs, thereby reducing memory fetches. In feedforward layers, weight stationary mapping is applied to ensure that large weight matrices are efficiently reused across computations. Additionally, these models incorporate fused attention kernels that integrate softmax and weighted summation into a single computation step, significantly enhancing execution speed (Jacobs et al. 2002).

For multilayer perceptrons, hybrid mapping strategies are employed to optimize performance through a combination of techniques that enhance both memory efficiency and computational throughput. Specifically, weight stationary execution is utilized to maximize the reuse of weights across activations, ensuring that these frequently accessed parameters remain readily available and reduce redundant memory accesses. In addition, blocked tiling strategies are implemented for large matrix multiplications, which significantly improve cache locality by partitioning the computation into manageable sub-blocks that fit within fast memory. Complementing these approaches, general matrix multiplication fusion is applied, effectively reducing memory stalls by merging consecutive matrix multiplication operations with subsequent functional transformations. Collectively, these optimizations illustrate how tailored mapping strategies can systematically balance memory constraints with computational demands in multilayer perceptron architectures.

Hybrid mapping strategies are widely employed in vision transformers, which seamlessly integrate convolutional and self-attention operations. In these models, the patch embedding layer performs a convolution-like operation that benefits from weight stationary mapping (Dosovitskiy et al. 2020). The self-attention layers, on the other hand, require activation stationary execution to efficiently reuse the key-value cache across multiple queries. Additionally, the MLP component leverages general matrix multiplication fusion and blocked tiling to execute dense matrix multiplications efficiently. This layer-specific optimization framework effectively balances memory locality with computational efficiency, rendering vision transformers particularly well-suited for AI accelerators.

11.6.4 Hybrid Strategies Hardware Implementations

Several modern AI accelerators incorporate hybrid mapping strategies to optimize execution by tailoring layer-specific techniques to the unique computational requirements of diverse neural network architectures. For example, Google TPUs employ weight stationary mapping for convolutional layers and activation stationary mapping for attention layers within transformer models, ensuring that the most critical data remains in fast memory. Likewise, NVIDIA GPUs leverage fused kernels alongside hybrid memory layouts, which enable the application of different mapping strategies within the same model to maximize performance. In addition, Graphcore IPUs dynamically select execution strategies on a per-layer basis to optimize memory access, thereby enhancing overall computational efficiency.

These real-world implementations illustrate how hybrid mapping strategies bridge the gap between different types of machine learning computations, ensuring that each layer executes with maximum efficiency. However, hardware support is essential for these techniques to be practical. Accelerators must provide architectural features such as programmable memory hierarchies, efficient interconnects, and specialized execution pipelines to fully exploit hybrid mapping.

Hybrid mapping provides a flexible and efficient approach to deep learning execution, enabling AI accelerators to adapt to the diverse computational requirements of modern architectures. By selecting the optimal mapping technique for each layer, hybrid strategies help reduce memory bandwidth constraints, improve data locality, and maximize parallelism.

While hybrid mapping strategies offer an effective way to optimize computations at a layer-specific level, they remain static design-time optimizations. In real-world AI workloads, execution conditions can change dynamically due to varying input sizes, memory contention, or hardware resource availability. Machine learning compilers and runtime systems extend these mapping techniques by introducing dynamic scheduling, memory optimizations, and automatic tuning mechanisms. These systems ensure that hybrid strategies are not just predefined execution choices, but rather adaptive mechanisms that allow deep learning workloads to operate efficiently across different accelerators and deployment environments. In the next section, we explore how machine learning compilers and runtime stacks enable these adaptive optimizations through just-in-time scheduling, memory-aware execution, and workload balancing strategies.

11.7 Compiler Support

The performance of machine learning acceleration depends not only on hardware capabilities but also on how efficiently models are translated into executable operations. The optimizations discussed earlier in this chapter, including kernel fusion, tiling, memory scheduling, and data movement strategies, are essential for maximizing efficiency. However, these optimizations must be systematically applied before execution to ensure they align with hardware constraints and computational requirements.

This process is handled by machine learning compilers, which form the software stack responsible for bridging high-level model representations with low-level hardware execution. The compiler optimizes the model by restructuring computations, selecting efficient execution kernels, and placing operations in a way that maximizes hardware utilization (0001 et al. 2018a).

While traditional compilers are designed for general-purpose computing, machine learning workloads require specialized approaches due to their reliance on tensor computations, parallel execution, and memory-intensive operations. To understand how these systems differ, we first compare machine learning compilers to their traditional counterparts.

11.7.1 ML vs Traditional Compilers

Machine learning workloads introduce unique challenges that traditional compilers were not designed to handle. Unlike conventional software execution, which primarily involves sequential or multi-threaded program flow, machine learning models are expressed as computation graphs that describe large-scale tensor operations. These graphs require specialized optimizations that traditional compilers cannot efficiently apply (Cui, Li, and Xie 2019).

Table 11.18 outlines the fundamental differences between traditional compilers and those designed for machine learning workloads. While traditional compilers optimize linear program execution through techniques like instruction scheduling and register allocation, ML compilers focus on optimizing computation graphs for efficient tensor operations. This distinction is critical, as ML compilers must incorporate domain-specific transformations such as kernel fusion, memory-aware scheduling, and hardware-accelerated execution plans to achieve high performance on specialized accelerators like GPUs and TPUs.

Table 11.18: Traditional vs. machine learning compilers and their optimization priorities.

Aspect	Traditional Compiler	Machine Learning Compiler
Input Representation	Linear program code (C, Python)	Computational graph (ML models)
Execution Model	Sequential or multi-threaded execution	Massively parallel tensor-based execution
Optimization Priorities	Instruction scheduling, loop unrolling, register allocation	Graph transformations, kernel fusion, memory-aware execution
Memory Management	Stack and heap memory allocation	Tensor layout transformations, tiling, memory-aware scheduling
Target Hardware	CPUs (general-purpose execution)	GPUs, TPUs, and custom accelerators
Compilation Output	CPU-specific machine code	Hardware-specific execution plan (kernels, memory scheduling)

This comparison highlights why machine learning models require a different compilation approach. Instead of optimizing instruction-level execution, machine learning compilers must transform entire computation graphs, apply tensor-aware memory optimizations, and schedule operations across thousands of parallel processing elements. These requirements make traditional compiler techniques insufficient for modern deep learning workloads.

11.7.2 ML Compilation Pipeline

Machine learning models, as defined in frameworks such as TensorFlow and PyTorch, are initially represented in a high-level computation graph that describes operations on tensors. However, these representations are not directly executable on hardware accelerators such as GPUs, TPUs, and custom AI chips. To achieve efficient execution, models must go through a compilation process that transforms them into optimized execution plans suited for the target hardware (Brain 2020).

The machine learning compilation workflow consists of several key stages, each responsible for applying specific optimizations that ensure minimal memory overhead, maximum parallel execution, and optimal compute utilization. These stages include:

1. **Graph Optimization**: The computation graph is restructured to eliminate inefficiencies.
2. **Kernel Selection**: Each operation is mapped to an optimized hardware-specific implementation.
3. **Memory Planning**: Tensor layouts and memory access patterns are optimized to reduce bandwidth consumption.
4. **Computation Scheduling**: Workloads are distributed across parallel processing elements to maximize hardware utilization.
5. **Code Generation**: The optimized execution plan is translated into machine-specific instructions for execution.

At each stage, the compiler applies theoretical optimizations discussed earlier, including kernel fusion, tiling, data movement strategies, and computation placement, ensuring that these optimizations are systematically incorporated into the final execution plan.

By understanding this workflow, we can see how machine learning acceleration is realized not just through hardware improvements but also through compiler-driven software optimizations.

11.7.3 Graph Optimization

AI accelerators provide specialized hardware to speed up computation, but raw model representations are not inherently optimized for execution on these accelerators. Machine learning frameworks define models using high-level computation graphs, where nodes represent operations (such as convolutions, matrix multiplications, and activations), and edges define data dependencies. However, if executed as defined, these graphs often contain redundant operations, inefficient memory access patterns, and suboptimal execution sequences that can prevent the hardware from operating at peak efficiency.

For example, in a Transformer model, the self-attention mechanism involves repeated accesses to the same key-value pairs across multiple attention heads. If compiled naïvely, the model may reload the same data multiple times, leading to excessive memory traffic (Shoeybi et al. 2019a). Similarly, in a Convolutional Neural Network (CNN), applying batch normalization and activation functions as separate operations after each convolution leads to unnecessary intermediate memory writes, increasing memory bandwidth usage. These inefficiencies are addressed during graph optimization, where the compiler restructures the computation graph to eliminate unnecessary operations and improve memory locality (0001 et al. 2018a).

The graph optimization phase of compilation is responsible for transforming this high-level computation graph into an optimized execution plan before it is mapped to hardware. Rather than requiring manual optimization, the compiler systematically applies transformations that improve data movement, reduce redundant computations, and restructure operations for efficient parallel execution (NVIDIA 2021).

At this stage, the compiler is still working at a hardware-agnostic level, focusing on high-level restructuring that improves efficiency before more hardware-specific optimizations are applied later.

11.7.3.1 Computation Graph Optimization

Graph optimization transforms the computation graph through a series of structured techniques designed to enhance execution efficiency. One key technique, which we discussed earlier, is kernel fusion, which merges consecutive operations to eliminate unnecessary memory writes and reduce the number of kernel launches. This approach is particularly effective in convolutional neural networks, where fusing convolution, batch normalization, and activation functions notably accelerates processing. Another important technique is computation reordering, which adjusts the execution order of operations to improve data locality and maximize parallel execution. For instance, in Transformer models, such reordering enables the reuse of cached key-value pairs rather than reloading them repeatedly from memory, thereby reducing latency.

Additionally, redundant computation elimination plays an important role. By identifying and removing duplicate or unnecessary operations, this method is especially beneficial in models with residual connections where common subexpressions might otherwise be redundantly computed. Furthermore, memory-aware dataflow adjustments enhance overall performance by refining tensor layouts and optimizing memory movement. For example, tiling matrix multiplications to meet the structural requirements of systolic arrays in TPUs ensures that hardware resources are utilized optimally. This combined approach not only reduces unnecessary processing but also aligns data storage and movement with the accelerator's strengths, leading to efficient execution across diverse AI workloads. Together, these techniques prepare the model for acceleration by minimizing overhead and ensuring an optimal balance between computational and memory resources.

11.7.3.2 AI Compilers Implementation

Modern AI compilers perform graph optimization through the use of automated pattern recognition and structured rewrite rules, systematically transforming computation graphs to maximize efficiency without manual intervention. For example, Google's XLA (Accelerated Linear Algebra) in TensorFlow applies graph-level transformations such as fusion and layout optimizations that streamline execution on TPUs and GPUs. Similarly, TVM (Tensor Virtual Machine) not only refines tensor layouts and adjusts computational structures but also tunes execution strategies across diverse hardware backends, which is particularly beneficial for deploying models on embedded Tiny ML devices with strict memory constraints.

NVIDIA's TensorRT, another specialized deep learning compiler, focuses on minimizing kernel launch overhead by fusing operations and optimizing execution scheduling on GPUs, thereby improving utilization and reducing inference latency in large-scale convolutional neural network applications. Additionally, MLIR (Multi-Level Intermediate Representation) facilitates flexible graph optimization across various AI accelerators by enabling multi-stage transformations that improve execution order and memory access patterns, thus easing the transition of models from CPU-based implementations to accelerator-optimized versions. These compilers preserve the mathematical integrity of the models while rewriting the computation graph to ensure that the subsequent hardware-specific optimizations can be effectively applied.

11.7.3.3 Graph Optimization Importance

Graph optimization enables AI accelerators to operate at peak efficiency. Without this phase, even the most optimized hardware would be underutilized, as models would be executed in a way that introduces unnecessary memory stalls, redundant computations, and inefficient data movement. By systematically restructuring computation graphs, the compiler arranges operations for efficient execution that mitigates bottlenecks before mapping to hardware, minimizes memory movement to keep tensors in high-speed memory, and optimizes parallel execution to reduce unnecessary serialization while enhancing hardware utilization. For instance, without proper graph optimization, a large Transformer model running on an edge device may experience excessive memory stalls due to suboptimal data access patterns; however, through effective graph restructuring, the model can operate with significantly reduced memory bandwidth consumption and latency, thus enabling real-time inference on devices with constrained resources.

With the computation graph now fully optimized, the next step in compilation is kernel selection, where the compiler determines which hardware-specific implementation should be used for each operation. This ensures that the structured execution plan is translated into optimized low-level instructions for the target accelerator.

11.7.4 Kernel Selection

At this stage, the compiler translates the abstract operations in the computation graph into optimized low-level functions, ensuring that execution is performed as efficiently as possible given the constraints of the target accelerator. A kernel is a specialized implementation of a computational operation designed to run efficiently on a particular hardware architecture. Most accelerators, including GPUs, TPUs, and custom AI chips, provide multiple kernel implementations for the same operation, each optimized for different execution scenarios. Choosing the right kernel for each operation is essential for maximizing computational throughput, minimizing memory stalls, and ensuring that the accelerator's specialized processing elements are fully utilized (NVIDIA 2021).

Kernel selection builds upon the graph optimization phase, ensuring that the structured execution plan is mapped to the most efficient implementation available. While graph optimization eliminates inefficiencies at the model level, kernel selection ensures that each individual operation is executed using the most efficient hardware-specific routine. The effectiveness of this process directly impacts the model's overall performance, as poor kernel choices can nullify the benefits of prior optimizations by introducing unnecessary computation overhead or memory bottlenecks (0001 et al. 2018a).

In a Transformer model, the matrix multiplications that dominate self-attention computations can be executed using different strategies depending on the available hardware. On a CPU, a general-purpose matrix multiplication routine is typically employed, exploiting vectorized execution to improve efficiency. In contrast, on a GPU, the compiler may select an implementation that leverages tensor cores to accelerate matrix multiplications using mixed-precision arithmetic. When the model is deployed on a TPU, the operation can be mapped onto a systolic array, ensuring that data flows through the accelerator in a manner that maximizes reuse and minimizes off-chip memory accesses. Additionally, for inference workloads, an integer arithmetic kernel may be preferable, as it facilitates computations in INT8 instead of floating-point precision, thereby reducing power consumption without significantly compromising accuracy.

In many cases, compilers do not generate custom kernels from scratch but instead select from vendor-optimized kernel libraries that provide highly tuned implementations for different architectures. For instance, cuDNN and cuBLAS offer optimized kernels for deep learning on NVIDIA GPUs, while oneDNN provides optimized execution for Intel architectures. Similarly, ACL (Arm Compute Library) is optimized for Arm-based devices, and Eigen and BLIS provide efficient CPU-based implementations of deep learning operations. These libraries allow the compiler to choose pre-optimized, high-performance kernels rather than having to reinvent execution strategies for each hardware platform.

11.7.4.1 Kernel Selection in AI Compilers

AI compilers use heuristics, profiling, and cost models to determine the best kernel for each operation. These strategies ensure that each computation is executed in a way that maximizes throughput and minimizes memory bottlenecks.

In rule-based selection, the compiler applies predefined heuristics based on the known capabilities of the hardware. For instance, XLA, the compiler used in TensorFlow, automatically selects tensor core-optimized kernels for NVIDIA GPUs when mixed-precision execution is enabled. These predefined rules allow the compiler to make fast, reliable decisions about which kernel to use without requiring extensive analysis.

Profile-guided selection takes a more dynamic approach, benchmarking different kernel options and choosing the one that performs best for a given workload. TVM, an open-source AI compiler, uses AutoTVM to empirically evaluate kernel performance, tuning execution strategies based on real-world execution times. By testing different kernels before deployment, profile-guided selection helps ensure that operations are assigned to the most efficient implementation under actual execution conditions.

Another approach, cost model-based selection, relies on performance predictions to estimate execution time and memory consumption for various kernels before choosing the most efficient one. MLIR, a compiler infrastructure designed for machine learning workloads, applies this technique to determine the most effective tiling and memory access strategies (Lattner et al. 2020). By modeling how different kernels interact with the accelerator's compute units and memory hierarchy, the compiler can select the kernel that minimizes execution cost while maximizing performance.

Many AI compilers also incorporate precision-aware kernel selection, where the selected kernel is optimized for specific numerical formats such as FP32, FP16, BF16, or INT8. Training workloads often prioritize higher precision (FP32, BF16) to maintain model accuracy, whereas inference workloads favor lower precision (FP16, INT8) to increase speed and reduce power consumption. For example, an NVIDIA GPU running inference with TensorRT can dynamically select FP16 or INT8 kernels based on a model's accuracy constraints. This trade-off between precision and performance is a key aspect of kernel selection, especially when deploying models in resource-constrained environments.

Some compilers go beyond static kernel selection and implement adaptive kernel tuning, where execution strategies are adjusted at runtime based on the system's workload and available resources. AutoTVM in TVM measures kernel performance across different workloads and dynamically refines execution strategies. TensorRT applies real-time optimizations based on batch size, memory constraints, and GPU load, adjusting kernel selection dynamically. Google's TPU compiler takes a similar approach, optimizing kernel selection based on cloud resource availability and execution environment constraints.

11.7.4.2 Kernel Selection Importance

The efficiency of AI acceleration depends not only on how computations are structured but also on how they are executed. Even the best-designed computation graph will fail to achieve peak performance if the selected kernels do not fully utilize the hardware's capabilities.

Proper kernel selection allows models to execute using the most efficient algorithms available for the given hardware, ensuring that memory is accessed in a way that avoids unnecessary stalls and that specialized acceleration features, such as tensor cores or systolic arrays, are leveraged wherever possible. Selecting an inappropriate kernel can lead to underutilized compute resources, excessive memory transfers, and increased power consumption, all of which limit the performance of AI accelerators.

For instance, if a Transformer model running on a GPU is assigned a non-tensor-core kernel for its matrix multiplications, it may execute at only a fraction of the possible performance. Conversely, if a model designed for FP32 execution is forced to run on an INT8-optimized kernel, it may experience significant numerical instability, degrading accuracy. These choices illustrate why kernel selection is as much about maintaining numerical correctness as it is about optimizing performance.

With kernel selection complete, the next stage in compilation involves execution scheduling and memory management, where the compiler determines how kernels are launched and how data is transferred between different levels of the memory hierarchy. These final steps in the compilation pipeline ensure that computations run with maximum parallelism while minimizing the overhead of data movement. As kernel selection determines what to execute, execution scheduling and memory management dictate when and how those kernels are executed, ensuring that AI accelerators operate at peak efficiency.

11.7.5 Memory Planning

The memory planning phase ensures that data is allocated and accessed in a way that minimizes memory bandwidth consumption, reduces latency, and maximizes cache efficiency (Y. Zhang, Li, and Ouyang 2020). Even with the most optimized execution plan, a model can still suffer from severe performance degradation if memory is not managed efficiently.

Machine learning workloads are often memory-intensive. They require frequent movement of large tensors between different levels of the memory hierarchy. The compiler must determine how tensors are stored, how they are accessed, and how intermediate results are handled to ensure that memory does not become a bottleneck.

The memory planning phase focuses on optimizing tensor layouts, memory access patterns, and buffer reuse to prevent unnecessary stalls and memory contention during execution. In this phase, tensors are arranged in a memory-efficient format that aligns with hardware access patterns, thereby minimizing the need for format conversions. Additionally, memory accesses are structured to reduce cache misses and stalls, which in turn lowers overall bandwidth consumption. Buffer reuse is also a critical aspect, as it reduces redundant memory allocations by intelligently managing intermediate results. Together, these strategies ensure that data is efficiently placed and accessed, thereby enhancing both computational performance and energy efficiency in AI workloads.

11.7.5.1 Memory Planning in AI Compilers

Memory planning is a complex problem because AI models must balance memory availability, reuse, and access efficiency while operating across multiple levels of the memory hierarchy. AI compilers use several key strategies to manage memory effectively and prevent unnecessary data movement.

The first step in memory planning is tensor layout optimization, where the compiler determines how tensors should be arranged in memory to maximize locality and prevent unnecessary data format conversions. Different hardware accelerators have different preferred storage layouts—for instance, NVIDIA

GPUs often use row-major storage (NHWC format), while TPUs favor channel-major layouts (NCHW format) to optimize memory coalescing (Martín Abadi, Agarwal, et al. 2016). The compiler automatically transforms tensor layouts based on the expected access patterns of the target hardware, ensuring that memory accesses are aligned for maximum efficiency.

Beyond layout optimization, memory planning also includes buffer allocation and reuse, where the compiler minimizes memory footprint by reusing intermediate storage whenever possible. Deep learning workloads generate many temporary tensors, such as activations and gradients, which can quickly overwhelm on-chip memory if not carefully managed. Instead of allocating new memory for each tensor, the compiler analyzes the computation graph to identify opportunities for buffer reuse, ensuring that intermediate values are stored and overwritten efficiently (G. A. Jones 2018).

Another critical aspect of memory planning is minimizing data movement between different levels of the memory hierarchy. AI accelerators typically have a mix of high-speed on-chip memory (such as caches or shared SRAM) and larger, but slower, external DRAM. If tensor data is repeatedly moved between these memory levels, the model may become memory-bound, reducing computational efficiency. To prevent this, compilers use tiling strategies that break large computations into smaller, memory-friendly chunks, allowing execution to fit within fast, local memory and reducing the need for costly off-chip memory accesses.

11.7.5.2 Memory Planning Importance

Without proper memory planning, even the most optimized computation graph and kernel selection will fail to deliver high performance. Excessive memory transfers, inefficient memory layouts, and redundant memory allocations can all lead to bottlenecks that prevent AI accelerators from reaching their peak throughput.

For instance, a CNN running on a GPU may achieve high computational efficiency in theory, but if its convolutional feature maps are stored in an incompatible format, for example, if it uses a row-major layout that necessitates conversion to a channel-friendly format such as NCHW or a variant like NHCW, constant tensor format conversions can introduce significant overhead. Similarly, a Transformer model deployed on an edge device may struggle to meet real-time inference requirements if memory is not carefully planned, leading to frequent off-chip memory accesses that increase latency and power consumption.

Through careful management of tensor placement, optimizing memory access patterns, and reducing unnecessary data movement, memory planning guarantees efficient operation of AI accelerators, leading to tangible performance improvements in real-world applications.

11.7.6 Computation Scheduling

With graph optimization completed, kernels selected, and memory planning finalized, the next step in the compilation pipeline is computation scheduling. This phase determines when and where each computation should be executed, ensuring that workloads are efficiently distributed across available processing elements while avoiding unnecessary stalls and resource contention (Rajbhandari et al. 2020; Zheng et al. 2020).

AI accelerators achieve high performance through massive parallelism, but without an effective scheduling strategy, computational units may sit idle, memory bandwidth may be underutilized, and execution efficiency may degrade. Computation scheduling is responsible for ensuring that all processing elements remain active, execution dependencies are managed correctly, and workloads are distributed optimally (Ziheng Jia et al. 2019).

In the scheduling phase, parallel execution, synchronization, and resource allocation are managed systematically. Task partitioning decomposes extensive computations into smaller, manageable tasks that can be distributed efficiently among multiple compute cores. Execution order optimization then determines the most effective sequence for launching these operations, maximizing hardware performance while reducing execution stalls. Additionally, resource allocation and synchronization are orchestrated to ensure that compute cores, memory bandwidth, and shared caches are utilized effectively, avoiding contention. Through these coordinated strategies, computation scheduling achieves optimal hardware utilization, minimizes memory access delays, and supports a streamlined and efficient execution process.

11.7.6.1 Computation Scheduling in AI Compilers

Computation scheduling is highly dependent on the underlying hardware architecture, as different AI accelerators have unique execution models that must be considered when determining how workloads are scheduled. AI compilers implement several key strategies to optimize scheduling for efficient execution.

One of the most fundamental aspects of scheduling is task partitioning, where the compiler divides large computational graphs into smaller, manageable units that can be executed in parallel. On GPUs, this typically means mapping matrix multiplications and convolutions to thousands of CUDA cores, while on TPUs, tasks are partitioned to fit within systolic arrays that operate on structured data flows (Norrie et al. 2021). In CPUs, partitioning is often focused on breaking computations into vectorized chunks that align with SIMD execution. The goal is to map workloads to available processing units efficiently, ensuring that each core remains active throughout execution.

In addition to task partitioning, scheduling also involves optimizing execution order to minimize dependencies and maximize throughput. Many AI models include operations that can be computed independently (e.g., different batches in a batch processing pipeline) alongside operations that have strict dependencies (e.g., recurrent layers in an RNN). AI compilers analyze these dependencies and attempt to rearrange execution where possible, reducing idle time and improving parallel efficiency. For example, in Transformer models, scheduling may prioritize preloading attention matrices into memory while earlier layers are still executing, ensuring that data is ready when needed (Shoeybi et al. 2019b).

Another crucial aspect of computation scheduling is resource allocation and synchronization, where the compiler determines how compute cores share memory and coordinate execution. Modern AI accelerators often support overlapping computation and data transfers, meaning that while one task executes, the next task can begin fetching its required data. Compilers take advantage of this by scheduling tasks in a way that hides memory latency, ensuring that execution remains compute-bound rather than memory-bound (0001 et al. 2018b). TensorRT and XLA, for example, employ streaming execution strategies where multiple kernels are launched in parallel, and synchronization is carefully managed to prevent execution stalls (Google, n.d.).

11.7.6.2 Computation Scheduling Importance

Without effective scheduling, even the most optimized model can suffer from underutilized compute resources, memory bottlenecks, and execution inefficiencies. Poor scheduling decisions can lead to idle processing elements, forcing expensive compute cores to wait for data or synchronization events before continuing execution.

For instance, a CNN running on a GPU may have highly optimized kernels and efficient memory layouts, but if its execution is not scheduled correctly, compute units may remain idle between kernel launches, reducing throughput. Similarly, a Transformer model deployed on a TPU may perform matrix multiplications efficiently but could experience performance degradation if attention layers are not scheduled to overlap efficiently with memory transfers.

Effective computation scheduling occupies a central role in the orchestration of parallel workloads, ensuring that processing elements are utilized to their fullest capacity while preventing idle cores—a critical aspect for maximizing overall throughput. By strategically overlapping computation with data movement, the scheduling mechanism effectively conceals memory latency, thereby preventing operational stalls during data retrieval. Moreover, by resolving execution dependencies with precision, it minimizes waiting periods and enhances the concurrent progression of computation and data transfer. This systematic integration of scheduling and data handling serves to not only elevate performance but also exemplify the rigorous engineering principles that underpin modern accelerator design.

11.7.6.3 Code Generation

Unlike the previous phases, which required AI-specific optimizations, code generation follows many of the same principles as traditional compilers. This process includes instruction selection, register allocation, and final optimization passes, ensuring that execution makes full use of hardware-specific features such as vectorized execution, memory prefetching, and instruction reordering.

For CPUs and GPUs, AI compilers typically generate machine code or optimized assembly instructions, while for TPUs, FPGAs, and other accelerators, the output may be optimized bytecode or execution graphs that are interpreted by the hardware's runtime system.

At this point, the compilation pipeline is complete: the original high-level model representation has been transformed into an optimized, executable format tailored for efficient execution on the target hardware. The combination of graph transformations, kernel selection, memory-aware execution, and parallel scheduling ensures that AI accelerators run workloads with maximum efficiency, minimal memory overhead, and optimal computational throughput.

11.7.7 Compilation-Runtime Support

The compiler plays a fundamental role in AI acceleration, transforming high-level machine learning models into optimized execution plans tailored to the constraints of specialized hardware. Throughout this section, we have seen how graph optimization restructures computation, kernel selection maps operations to hardware-efficient implementations, memory planning optimizes data placement, and computation scheduling ensures efficient parallel execution. Each of these phases is crucial in enabling AI models to fully leverage modern accelerators, ensuring high throughput, minimal memory overhead, and efficient execution pipelines.

However, compilation alone is not enough to guarantee efficient execution in real-world AI workloads. While compilers statically optimize computation based on known model structures and hardware capabilities, AI execution environments are often dynamic and unpredictable. Batch sizes fluctuate, hardware resources may be shared across multiple workloads, and accelerators must adapt to real-time performance constraints. In these cases, a static execution plan is insufficient, and runtime management becomes critical in ensuring that models execute optimally under real-world conditions.

This transition from static compilation to adaptive execution is where AI runtimes come into play. Runtimes provide dynamic memory allocation, real-time kernel selection, workload scheduling, and multi-chip coordination, allowing AI models to adapt to varying execution conditions while maintaining efficiency. In the next section, we explore how AI runtimes extend the capabilities of compilers, enabling models to run effectively in diverse and scalable deployment scenarios.

11.8 Runtime Support

While compilers optimize AI models before execution, real-world deployment introduces dynamic and unpredictable conditions that static compilation alone cannot fully address (NVIDIA 2021). AI workloads operate in varied execution environments, where factors such as fluctuating batch sizes, shared hardware resources, memory contention, and latency constraints necessitate real-time adaptation. Precompiled execution plans, optimized for a fixed set of assumptions, may become suboptimal when actual runtime conditions change.

To bridge this gap, AI runtimes provide a dynamic layer of execution management, extending the optimizations performed at compile time with real-time decision-making. Unlike traditional compiled programs that execute a fixed sequence of instructions, AI workloads require adaptive control over memory allocation, kernel execution, and resource scheduling. AI runtimes continuously monitor execution conditions and make on-the-fly adjustments to ensure that machine learning models fully utilize available hardware while maintaining efficiency and performance guarantees.

At a high level, AI runtimes manage three critical aspects of execution:

1. **Kernel Execution Management**: AI runtimes dynamically select and dispatch computation kernels based on the current system state, ensuring that workloads are executed with minimal latency.
2. **Memory Adaptation and Allocation**: Since AI workloads frequently process large tensors with varying memory footprints, runtimes adjust memory allocation dynamically to prevent bottlenecks and excessive data movement (Y. Huang et al. 2019).
3. **Execution Scaling**: AI runtimes handle workload distribution across multiple accelerators, supporting large-scale execution in multi-chip, multi-node, or cloud environments (Mirhoseini et al. 2017).

By dynamically handling these execution aspects, AI runtimes complement compiler-based optimizations, ensuring that models continue to perform efficiently under varying runtime conditions. The next section explores how AI runtimes differ from traditional software runtimes, highlighting why machine learning workloads require fundamentally different execution strategies compared to conventional CPU-based programs.

11.8.1 ML vs Traditional Runtimes

Traditional software runtimes are designed for managing general-purpose program execution, primarily handling sequential and multi-threaded workloads on CPUs. These runtimes allocate memory, schedule tasks, and optimize execution at the level of individual function calls and instructions. In contrast, AI runtimes are specialized for machine learning workloads, which require massively parallel computation, large-scale tensor operations, and dynamic memory management.

Table 11.19 highlights the fundamental differences between traditional and AI runtimes. One of the key distinctions lies in execution flow. Traditional software runtimes operate on a predictable, structured execution model where function calls and CPU threads follow a predefined control path. AI runtimes, however, execute computational graphs, requiring complex scheduling decisions that account for dependencies between tensor operations, parallel kernel execution, and efficient memory access.

Table 11.19: Key differences between traditional and AI runtimes.

Aspect	Traditional Runtime	AI Runtime
Execution Model	Sequential or multi-threaded execution	Massively parallel tensor execution
Task Scheduling	CPU thread management	Kernel dispatch across accelerators
Memory Management	Static allocation (stack/heap)	Dynamic tensor allocation, buffer reuse
Optimization Priorities	Low-latency instruction execution	Minimizing memory stalls, maximizing parallel execution
Adaptability	Mostly static execution plan	Adapts to batch size and hardware availability
Target Hardware	CPUs (general-purpose execution)	GPUs, TPUs, and custom accelerators

Memory management is another major differentiator. Traditional software runtimes handle small, frequent memory allocations, optimizing for cache efficiency and low-latency access. AI runtimes, in contrast, must dynamically allocate, reuse, and optimize large tensors, ensuring that memory access patterns align with accelerator-friendly execution. Poor memory management in AI workloads can lead to performance bottlenecks, particularly due to excessive off-chip memory transfers and inefficient cache usage.

Moreover, AI runtimes are inherently designed for adaptability. While traditional runtimes often follow a mostly static execution plan, AI workloads typically operate in highly variable execution environments, such as cloud-based accelerators or multi-tenant hardware. As a result, AI runtimes must continuously adjust batch sizes, reallocate compute resources, and manage real-time scheduling decisions to maintain high throughput and minimize execution delays.

These distinctions demonstrate why AI runtimes require fundamentally different execution strategies compared to traditional software runtimes. Rather than simply managing CPU processes, AI runtimes must oversee large-scale tensor execution, multi-device coordination, and real-time workload adaptation to ensure that machine learning models can run efficiently under diverse and ever-changing deployment conditions.

11.8.2 Dynamic Kernel Execution

Dynamic kernel execution is the process of mapping machine learning models to hardware and optimizing runtime execution. While static compilation provides a solid foundation, efficient execution of machine learning workloads requires real-time adaptation to fluctuating conditions such as available memory, data sizes, and computational loads. The runtime functions as an intermediary that continuously adjusts execution strategies to match both the constraints of the underlying hardware and the characteristics of the workload.

When mapping a machine learning model to hardware, individual computational operations, including matrix multiplications, convolutions, and activation functions, must be assigned to the most appropriate processing units. This mapping is not fixed; it must be modified during runtime in response to changes in input data, memory availability, and overall system load. Dynamic kernel execution allows the runtime to make real-time decisions regarding kernel selection, execution order, and memory management, ensuring that workloads remain efficient despite these changing conditions.

For example, consider an AI accelerator executing a deep neural network (DNN) for image classification. If an incoming batch of high-resolution images requires significantly more memory than expected, a statically planned execution may cause cache thrashing or excessive off-chip memory accesses. Instead, a dynamic runtime can adjust tiling strategies on the fly, breaking down tensor operations into smaller tiles that fit within the high-speed on-chip memory. This prevents memory stalls and ensures optimal utilization of caches.

Similarly, when running a transformer-based natural language processing (NLP) model, the sequence length of input text may vary between inference requests. A static execution plan optimized for a fixed sequence length may lead to underutilization of compute resources when processing shorter sequences or excessive memory pressure with longer sequences. Dynamic kernel execution can mitigate this by selecting different kernel implementations based on the actual sequence length, dynamically adjusting memory allocations and execution strategies to maintain efficiency.

Moreover, overlapping computation with memory movement is a vital strategy to mitigate performance bottlenecks. AI workloads often encounter delays due to memory-bound issues, where data movement between memory hierarchies limits computation speed. To combat this, AI runtimes implement techniques like asynchronous execution and double buffering, ensuring that computations proceed without waiting for memory transfers to complete. In a large-scale model, for instance, image data can be prefetched while computations are performed on the previous batch, thus maintaining a steady flow of data and avoiding pipeline stalls.

Another practical example is the execution of convolutional layers in a CNN on a GPU. If multiple convolution kernels need to be scheduled, a static scheduling approach may lead to inefficient resource utilization due to variation in layer sizes and compute requirements. By dynamically scheduling kernel execution, AI runtimes can prioritize smaller kernels when compute units are partially occupied, improving hardware utilization. For instance, in NVIDIA's TensorRT runtime, fusion of small kernels into larger execution units is done dynamically to avoid launch overhead, optimizing latency-sensitive inference tasks.

Dynamic kernel execution plays an essential role in ensuring that machine learning models are executed efficiently. By dynamically adjusting execution strategies in response to real-time system conditions, AI runtimes optimize both training and inference performance across various hardware platforms.

11.8.3 Runtime Kernel Selection

While compilers may perform an initial selection of kernels based on static analysis of the machine learning model and hardware target, AI runtimes often need to override these decisions during execution. Real-time factors, such as available memory, hardware utilization, and workload priorities, may differ significantly from the assumptions made during compilation. By dynamically selecting and switching kernels at runtime, AI runtimes can adapt to these changing conditions, ensuring that models continue to perform efficiently.

For instance, consider transformer-based language models, where a significant portion of execution time is spent on matrix multiplications. The AI runtime must determine the most efficient way to execute these operations based on the current system state. If the model is running on a GPU with specialized Tensor Cores, the runtime may switch from a standard FP32 kernel to an FP16 kernel to take advantage of hardware acceleration (Shoeybi et al. 2019a). Conversely, if the lower precision of FP16 causes unacceptable numerical instability, the runtime can opt for mixed-precision execution, selectively using FP32 where higher precision is necessary.

Memory constraints also influence kernel selection. When memory bandwidth is limited, the runtime may adjust its execution strategy, reordering operations or changing the tiling strategy to fit computations into the available cache rather than relying on slower main memory. For example, a large matrix multiplication may be broken into smaller chunks, ensuring that the computation fits into the on-chip memory of the GPU, reducing overall latency.

Additionally, batch size can influence kernel selection. For workloads that handle a mix of small and large batches, the AI runtime may choose a latency-optimized kernel for small batches and a throughput-optimized kernel for large-scale batch processing. This adjustment ensures that the model continues to operate efficiently across different execution scenarios, without the need for manual tuning.

11.8.4 Kernel Scheduling and Utilization

Once the AI runtime selects an appropriate kernel, the next step is scheduling it in a way that maximizes parallelism and resource utilization. Unlike traditional task schedulers, which are designed to manage CPU threads, AI runtimes must coordinate a much larger number of tasks across parallel execution units such as GPU cores, tensor processing units, or custom AI accelerators (Jouppi et al. 2017). Effective scheduling ensures that these computational resources are kept fully engaged, preventing bottlenecks and maximizing throughput.

For example, in image recognition models that use convolutional layers, operations can be distributed across multiple processing units, enabling different filters to run concurrently. This parallelization ensures that the available hardware is fully utilized, speeding up execution. Similarly, batch normalization and activation functions must be scheduled efficiently to avoid unnecessary delays. If these operations are not interleaved with other computations, they may block the pipeline and reduce overall throughput.

Efficient kernel scheduling can also be influenced by real-time memory management . AI runtimes ensure that intermediate data, such as feature maps in deep neural networks, are preloaded into cache before they are needed. This proactive management helps prevent delays caused by waiting for data to be loaded from slower memory tiers, ensuring continuous execution.

These techniques enable AI runtimes to ensure optimal resource utilization and efficient parallel computation, which are essential for the high-performance execution of machine learning models, particularly in environments that require scaling across multiple hardware accelerators.

11.9 Multi-Chip AI Acceleration

Modern AI workloads increasingly demand computational resources that exceed the capabilities of single-chip accelerators. This section examines how AI systems scale from individual processors to multi-chip architectures, analyzing the motivation behind different scaling approaches and their impact on system design. By understanding this progression, we can better appreciate how each component of the AI hardware stack, ranging from compute units to memory systems, must adapt to support large-scale machine learning workloads.

The scaling of AI systems follows a natural progression, starting with integration within a single package through chiplet architectures, extending to multi-GPU configurations within a server, expanding to distributed accelerator pods, and culminating in wafer-scale integration. Each approach presents unique trade-offs between computational density, communication overhead, and system complexity. For instance, chiplet architectures maintain high-speed interconnects within a package, while distributed systems sacrifice communication latency for massive parallelism.

Understanding these scaling strategies is essential for several reasons. First, it provides insight into how different hardware architectures address the growing computational demands of AI workloads. Second, it reveals the fundamental challenges that arise when extending beyond single-chip execution, such as managing inter-chip communication and coordinating distributed computation. Finally, it establishes the foundation for subsequent discussions on how mapping strategies, compilation techniques, and runtime systems evolve to support efficient execution at scale.

11.9.0.1 Chiplet-Based Architectures

The first step in scaling AI accelerators is to move beyond a single monolithic chip while still maintaining a compact, tightly integrated design. Chiplet architectures achieve this by partitioning large designs into smaller, modular dies that are interconnected within a single package, as illustrated in Figure 11.9.

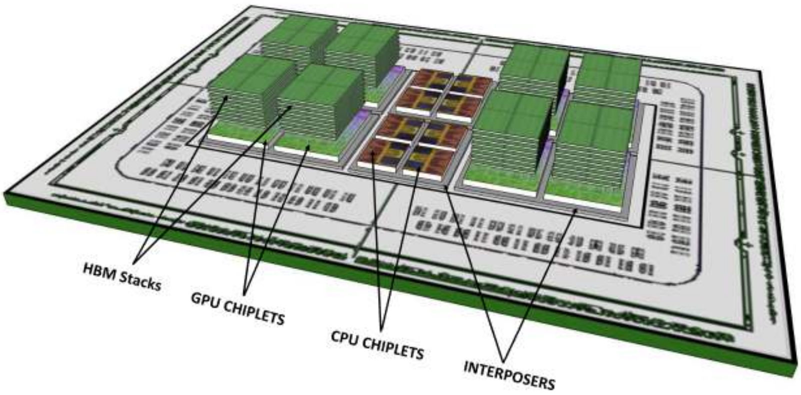


Figure 11.9: AMD's chiplet-based architecture.

Modern AI accelerators, such as AMD's Instinct MI300, take this approach by integrating multiple compute chiplets alongside memory chiplets, linked by high-speed die-to-die interconnects (Kannan, Dubey, and Horowitz 2023). This modular design allows manufacturers to bypass the manufacturing limits of monolithic chips while still achieving high-density compute.

However, even within a single package, scaling is not without challenges. Inter-chiplet communication latency, memory coherence, and thermal management become critical factors as more chiplets are integrated. Unlike traditional multi-chip systems, chiplet-based designs must carefully balance latency-sensitive workloads across multiple dies without introducing excessive bottlenecks.

11.9.0.2 Multi-GPU Systems

Beyond chiplet-based designs, AI workloads often require multiple discrete GPUs working together. In multi-GPU systems, each accelerator has its own dedicated memory and compute resources, but they must efficiently share data and synchronize execution.

A common example is NVIDIA DGX systems, which integrate multiple GPUs connected via NVLink[8] or PCIe[9]. This architecture enables workloads to be split across GPUs, typically using data parallelism (where each GPU processes a different batch of data) or model parallelism (where different GPUs handle different parts of a neural network) (Ben-Nun and Hoefler 2019).

As illustrated in Figure 11.10, NVSwitch interconnects enable high-speed communication between GPUs, reducing bottlenecks in distributed training. However, scaling up the number of GPUs introduces new challenges. Cross-GPU communication bandwidth, memory consistency, and workload scheduling become critical constraints, particularly for large-scale models requiring frequent data exchanges. Unlike chiplets, which leverage high-speed die-to-die interconnects, discrete GPUs rely on external links, incurring higher latency and synchronization overhead.

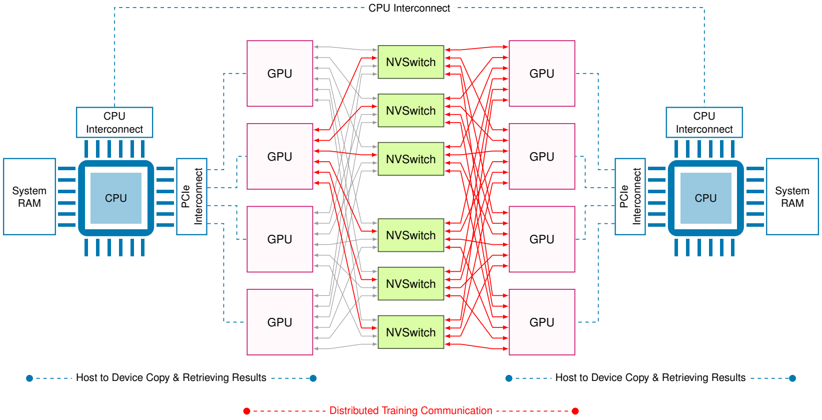


Figure 11.10: Multi-GPU architecture with NVSwitch interconnects.

11.9.0.3 TPU Pods

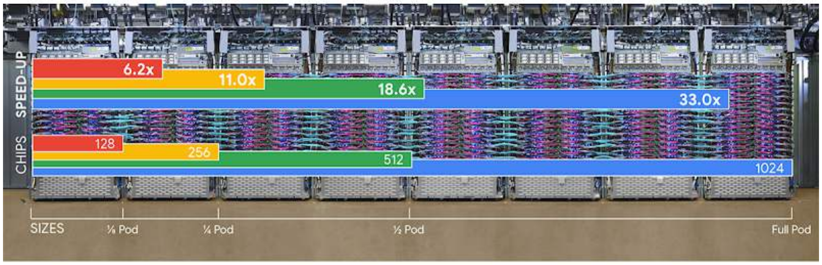
As models and datasets continue to expand, training and inference workloads must extend beyond single-server configurations. This scaling requirement has led to the development of sophisticated distributed systems where multiple accelerators communicate across networks. Google's TPU Pods represent a pioneering approach to this challenge, interconnecting hundreds of TPUs to function as a unified system (Jouppi et al. 2020).

The architectural design of TPU Pods differs fundamentally from traditional multi-GPU systems. While multi-GPU configurations typically rely on NVLink or PCIe connections within a single machine, TPU Pods employ high-bandwidth optical links to interconnect accelerators at data center scale. This design implements a 2D torus interconnect topology, enabling efficient data exchange between accelerators while minimizing communication bottlenecks as workloads scale across nodes.

[8] NVLink: A high-speed interconnect that enables faster data transfers between GPUs, reducing communication bottlenecks.

[9] PCIe (Peripheral Component Interconnect Express): A common interface for connecting high-speed components; however, it typically offers lower bandwidth compared to NVLink for GPU-to-GPU communication.

The effectiveness of this architecture is demonstrated in its performance scaling capabilities. As illustrated in Figure 11.11, TPU Pod performance exhibits near-linear scaling when running ResNet-50, from quarter-pod to full-pod configurations. The system achieves a remarkable 33.0x speedup when scaled to 1024 chips compared to a 16-TPU baseline. This scaling efficiency is particularly noteworthy in larger configurations, where performance continues to scale strongly even as the system expands from 128 to 1024 chips.



Cloud TPU v3 Pod performance scaling on ResNet-50 across a range of slice sizes relative to a 16-TPU-chip baseline [1,3]

Figure 11.11: Cloud TPU v3 pods and their performance on ResNet-50 across a range of slice sizes relative to a 16-TPU-chip baseline.

However, distributing AI workloads across an entire data center introduces unique challenges. Systems must contend with interconnect congestion, synchronization delays, and the complexities of efficient workload partitioning. Unlike multi-GPU setups where accelerators share memory hierarchies, TPU Pods operate in a fully distributed memory system. This architecture necessitates explicit communication strategies to manage data movement effectively, requiring careful consideration of data placement and transfer patterns to maintain scaling efficiency.

11.9.0.4 Wafer-Scale AI

At the frontier of AI scaling, wafer-scale integration represents a paradigm shift—abandoning traditional multi-chip architectures in favor of a single, massive AI processor. Rather than partitioning computation across discrete chips, this approach treats an entire silicon wafer as a unified compute fabric, eliminating the inefficiencies of inter-chip communication.

As shown in Figure 11.12, Cerebras' Wafer-Scale Engine (WSE) processors break away from the historical transistor scaling trends of CPUs, GPUs, and TPUs. While these architectures have steadily increased transistor counts along an exponential trajectory, WSE introduces an entirely new scaling paradigm, integrating trillions of transistors onto a single wafer—far surpassing even the most advanced GPUs and TPUs. With WSE-3, this trajectory continues, pushing wafer-scale AI to unprecedented levels (Systems 2021a).

The fundamental advantage of wafer-scale AI is its ultra-fast, on-die communication. Unlike chiplets, GPUs, or TPU Pods, where data must traverse physical boundaries between separate devices, wafer-scale AI enables near-instantaneous data transfer across its vast compute array. This architecture drastically reduces communication latency, unlocking performance levels that are unachievable with conventional multi-chip systems.

However, achieving this level of integration introduces formidable engineering challenges. Thermal dissipation, fault tolerance, and manufacturing yield become major constraints when fabricating a processor of this scale. Unlike distributed TPU systems, which mitigate failures by dynamically re-routing workloads, wafer-scale AI must incorporate built-in redundancy mechanisms to tolerate localized defects in the silicon. Successfully addressing these challenges is essential to realizing the full potential of wafer-scale computing as the next frontier in AI acceleration.

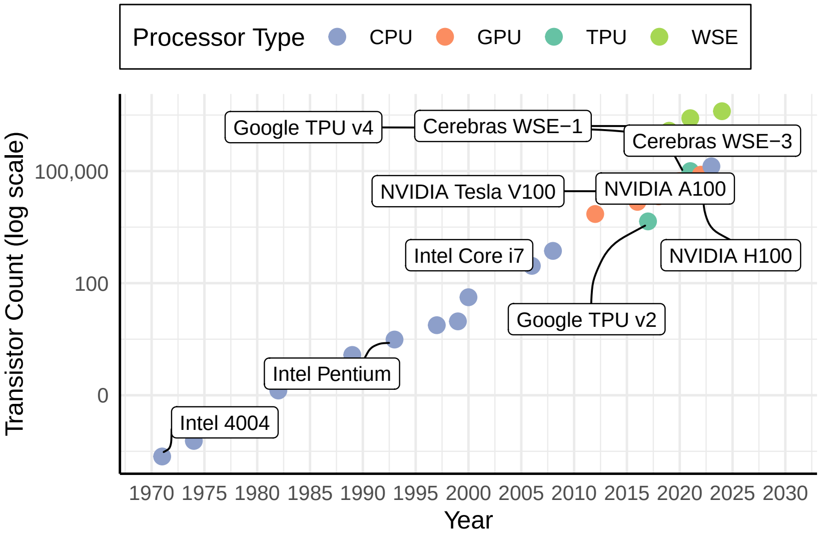


Figure 11.12: Processor transistor count over time.

11.9.0.5 AI Systems Scaling Trajectory

Table 11.20 illustrates the progressive scaling of AI acceleration, from single-chip processors to increasingly complex architectures such as chiplet-based designs, multi-GPU systems, TPU Pods, and wafer-scale AI. Each step in this evolution introduces new challenges related to data movement, memory access, interconnect efficiency, and workload distribution. While chiplets enable modular scaling within a package, they introduce latency and memory coherence issues. Multi-GPU systems rely on high-speed interconnects like NVLink but face synchronization and communication bottlenecks. TPU Pods push scalability further by distributing workloads across clusters, yet they must contend with interconnect congestion and workload partitioning. At the extreme end, wafer-scale AI integrates an entire wafer into a single computational unit, presenting unique challenges in thermal management and fault tolerance.

Table 11.20: Scaling trajectory of AI systems and associated challenges.

Scaling Approach	Key Feature	Challenges
Chiplets	Modular scaling within a package	Inter-chiplet latency, memory coherence
Multi-GPU	External GPU interconnects (NVLink)	Synchronization overhead, communication bottlenecks
TPU Pods	Distributed accelerator clusters	Interconnect congestion, workload partitioning
Wafer-Scale AI	Entire wafer as a single processor	Thermal dissipation, fault tolerance

11.9.1 Computation and Memory Scaling Changes

As AI systems scale from single-chip accelerators to multi-chip architectures, the fundamental challenges in computation and memory evolve. In a single accelerator, execution is primarily optimized for locality—ensuring that computations are mapped efficiently to available processing elements while minimizing memory access latency. However, as AI systems extend beyond a single chip, the scope of these optimizations expands significantly. Computation must now be distributed across multiple accelerators, and memory access patterns become constrained by interconnect bandwidth and communication overhead.

11.9.1.1 Multi-chip Execution Mapping

In single-chip AI accelerators, computation placement is concerned with mapping workloads to PEs, vector units, and tensor cores. Mapping strategies aim to maximize data locality, ensuring that computations access nearby memory to reduce costly data movement.

As AI systems scale to multi-chip execution, computation placement must consider several critical factors. Workloads need to be partitioned across multiple accelerators, which requires explicit coordination of execution order and dependencies. This division is essential due to the inherent latency associated with cross-chip communication, which contrasts sharply with single-chip systems that benefit from shared on-chip memory. Accordingly, computation scheduling must be interconnect-aware to manage these delays effectively. Additionally, achieving load balancing across accelerators is vital; an uneven distribution of tasks can result in some accelerators remaining underutilized while others operate at full capacity, ultimately hindering overall system performance.

For example, in multi-GPU training, computation mapping must ensure that each GPU has a balanced portion of the workload while minimizing expensive cross-GPU communication. Similarly, in TPU Pods, mapping strategies must align with the torus interconnect topology, ensuring that computation is placed to minimize long-distance data transfers.

Thus, while computation placement in single-chip systems is a local optimization problem, in multi-chip architectures, it becomes a global optimization challenge where execution efficiency depends on minimizing inter-chip communication and balancing workload distribution.

11.9.1.2 Distributed Access Memory Allocation

Memory allocation strategies in single-chip AI accelerators are designed to minimize off-chip memory accesses by leveraging on-chip caches, SRAM, and

HBM. Techniques such as tiling, data reuse, and kernel fusion ensure that computations make efficient use of fast local memory.

In multi-chip AI systems, each accelerator manages its own local memory, which necessitates the explicit allocation of model parameters, activations, and intermediate data across the devices. Unlike single-chip execution where data is fetched once and reused, multi-chip setups require deliberate strategies to minimize redundant data transfers, as data must be communicated between accelerators. Additionally, when overlapping data is processed by multiple accelerators, the synchronization of shared data can introduce significant overhead that must be carefully managed to ensure efficient execution.

For instance, in multi-GPU deep learning, gradient synchronization across GPUs is a memory-intensive operation that must be optimized to avoid network congestion (Shallue, Lee, et al. 2019). In wafer-scale AI, memory allocation must account for fault tolerance and redundancy mechanisms, ensuring that defective regions of the wafer do not disrupt execution.

Thus, while memory allocation in single-chip accelerators focuses on local cache efficiency, in multi-chip architectures, it must be explicitly coordinated across accelerators to balance memory bandwidth, minimize redundant transfers, and reduce synchronization overhead.

11.9.1.3 Data Movement Constraints

In single-chip AI accelerators, data movement optimization is largely focused on minimizing on-chip memory access latency. Techniques such as weight stationarity, input stationarity, and tiling ensure that frequently used data remains close to the execution units, reducing off-chip memory traffic.

In multi-chip architectures, data movement transcends being merely an intra-chip issue and becomes a significant system-wide bottleneck. Scaling introduces several critical challenges, foremost among them being inter-chip bandwidth constraints; communication links such as PCIe, NVLink, and TPU interconnects operate at speeds that are considerably slower than those of on-chip memory accesses. Additionally, when accelerators share model parameters or intermediate computations, the resulting data synchronization overhead, which encompass latency and contention, can markedly impede execution. Finally, optimizing collective communication is essential for workloads that require frequent data exchanges, such as gradient updates in deep learning training, where minimizing synchronization penalties is imperative for achieving efficient system performance.

For example, in TPU Pods, systolic execution models ensure that data moves in structured patterns, reducing unnecessary off-chip transfers. In multi-GPU inference, techniques like asynchronous data fetching and overlapping computation with communication help mitigate inter-chip latency.

Thus, while data movement optimization in single-chip systems focuses on cache locality and tiling, in multi-chip architectures, the primary challenge is reducing inter-chip communication overhead to maximize efficiency.

11.9.1.4 Compilers and Runtimes Adaptation

As AI acceleration extends beyond a single chip, compilers and runtimes must adapt to manage computation placement, memory organization, and execution scheduling across multiple accelerators. The fundamental principles of locality, parallelism, and efficient scheduling remain essential, but their implementation requires new strategies for distributed execution.

One of the primary challenges in scaling AI execution is computation placement. In a single-chip accelerator, workloads are mapped to processing elements, vector units, and tensor cores with an emphasis on minimizing on-chip data movement and maximizing parallel execution. However, in a multi-chip system, computation must be partitioned hierarchically, where workloads are distributed not just across cores within a chip, but also across multiple accelerators. Compilers handle this by implementing interconnect-aware scheduling, optimizing workload placement to minimize costly inter-chip communication.

Similarly, memory management evolves as scaling extends beyond a single accelerator. In a single-chip system, local caching, HBM reuse, and efficient tiling strategies ensure that frequently accessed data remains close to computation units. However, in a multi-chip system, each accelerator has its own independent memory, requiring explicit memory partitioning and coordination. Compilers optimize memory layouts for distributed execution, while runtimes introduce data prefetching and caching mechanisms to reduce inter-chip memory access overhead.

Beyond computation and memory, data movement becomes a major bottleneck at scale. In a single-chip accelerator, efficient on-chip caching and minimized DRAM accesses ensure that data is reused efficiently. However, in a multi-chip system, communication-aware execution becomes critical, requiring compilers to generate execution plans that overlap computation with data transfers. Runtimes handle inter-chip synchronization, ensuring that workloads are not stalled by waiting for data to arrive from remote accelerators.

Finally, execution scheduling must be extended for global coordination. In single-chip AI execution, scheduling is primarily concerned with parallelism and maximizing compute occupancy within the accelerator. However, in a multi-chip system, scheduling must balance workload distribution across accelerators while taking interconnect bandwidth and synchronization latency into account. Runtimes manage this complexity by implementing adaptive scheduling strategies that dynamically adjust execution plans based on system state and network congestion.

Table 11.21 summarizes these key adaptations, highlighting how compilers and runtimes extend their capabilities to efficiently support multi-chip AI execution.

Thus, while the fundamentals of AI acceleration remain intact, compilers and runtimes must extend their functionality to operate efficiently across distributed systems. The next section will explore how mapping strategies evolve to further optimize multi-chip AI execution.

Table 11.21: Adaptations in computation placement, memory management, and scheduling for multi-chip AI execution.

Aspect	Single-Chip AI Accelerator	Multi-Chip AI System & How Compilers/Runtimes Adapt
Computation Placement	Local PEs, tensor cores, vector units	Hierarchical mapping, interconnect-aware scheduling
Memory Management	Caching, HBM reuse, local tiling	Distributed allocation, prefetching, caching
Data Movement	On-chip reuse, minimal DRAM access	Communication-aware execution, overlap transfers
Execution Scheduling	Parallelism, compute occupancy	Global scheduling, interconnect-aware balancing

11.9.2 Execution Models Adaptation

As AI accelerators scale beyond a single chip, execution models must evolve to account for the complexities introduced by distributed computation, memory partitioning, and inter-chip communication. In single-chip accelerators, execution is optimized for local processing elements, with scheduling strategies that balance parallelism, locality, and data reuse. However, in multi-chip AI systems, execution must now be coordinated across multiple accelerators, introducing new challenges in workload scheduling, memory coherence, and interconnect-aware execution.

This section explores how execution models change as AI acceleration scales, focusing on scheduling, memory coordination, and runtime management in multi-chip systems.

11.9.2.1 Cross-Accelerator Scheduling

In single-chip AI accelerators, execution scheduling is primarily aimed at optimizing parallelism within the processor. This involves ensuring that workloads are effectively mapped to tensor cores, vector units, and special function units by employing techniques designed to enhance data locality and resource utilization. For instance, static scheduling uses a predetermined execution order that is carefully optimized for locality and reuse, while dynamic scheduling adapts in real time to variations in workload demands. Additionally, pipeline execution divides computations into stages, thereby maximizing hardware utilization by maintaining a continuous flow of operations.

In contrast, scheduling in multi-chip architectures must address the additional challenges posed by inter-chip dependencies. Workload partitioning in such systems involves distributing tasks across various accelerators such that each receives an optimal share of the workload, all while minimizing the overhead caused by excessive communication. Moreover, interconnect-aware scheduling is essential to align execution timing with the constraints of inter-chip bandwidth, thus preventing performance stalls. Latency hiding techniques also play a critical role, as they enable the overlapping of computation with communication, effectively reducing waiting times.

For example, in multi-GPU inference scenarios, execution scheduling is implemented in a way that allows data to be prefetched concurrently with computation, thereby mitigating memory stalls. Similarly, TPU Pods leverage the systolic array model to tightly couple execution scheduling with data flow, ensuring that each TPU core receives its required data precisely when needed. Therefore, while single-chip execution scheduling is focused largely on maximizing internal parallelism, multi-chip systems require a more holistic approach that explicitly manages communication overhead and synchronizes workload distribution across accelerators.

11.9.2.2 Cross-Accelerator Coordination

In single-chip AI accelerators, memory coordination is managed through sophisticated local caching strategies that keep frequently used data in close proximity to the execution units. Techniques such as tiling, kernel fusion, and data reuse are employed to reduce the dependency on slower memory hierarchies, thereby enhancing performance and reducing latency.

In contrast, multi-chip architectures present a distributed memory coordination challenge that necessitates more deliberate management. Each accelerator in such a system possesses its own independent memory, which must be organized through explicit memory partitioning to minimize cross-chip data accesses. Additionally, ensuring consistency and synchronization of shared data across accelerators is essential to maintain computational correctness. Efficient communication mechanisms must also be implemented to schedule data transfers in a way that limits overhead associated with synchronization delays.

For instance, in distributed deep learning training, model parameters must be synchronized across multiple GPUs using methods such as all-reduce, where gradients are aggregated across accelerators while reducing communication latency. In wafer-scale AI, memory coordination must further address fault-tolerant execution, ensuring that defective areas do not compromise overall system performance. Consequently, while memory coordination in single-chip systems is primarily concerned with cache optimization, multi-chip architectures require comprehensive management of distributed memory access, synchronization, and communication to achieve efficient execution.

11.9.2.3 Cross-Accelerator Execution Management

Execution in single-chip AI accelerators is managed by AI runtimes that handle workload scheduling, memory allocation, and hardware execution. These runtimes optimize execution at the kernel level, ensuring that computations are executed efficiently within the available resources.

In multi-chip AI systems, runtimes must incorporate a comprehensive strategy for distributed execution orchestration. This approach ensures that both computation and memory access are seamlessly coordinated across multiple accelerators, enabling efficient utilization of hardware resources and minimizing bottlenecks associated with data transfers.

Furthermore, these systems require robust mechanisms for cross-chip workload synchronization. Careful management of dependencies and timely coordination between accelerators are essential to prevent stalls in execution that may arise from delays in inter-chip communication. Such synchronization is critical for maintaining the flow of computation, particularly in environments where latency can significantly impact overall performance.

Finally, adaptive execution models play a pivotal role in contemporary multi-chip architectures. These models dynamically adjust execution plans based on current hardware availability and communication constraints, ensuring that the system can respond to changing conditions and optimize performance in real time. Together, these strategies provide a resilient framework for managing the complexities of distributed AI execution.

For example, in Google's TPU Pods, the TPU runtime is responsible for scheduling computations across multiple TPU cores, ensuring that workloads are executed in a way that minimizes communication bottlenecks. In multi-GPU frameworks like PyTorch and TensorFlow, runtime execution must synchronize operations across GPUs, ensuring that data is transferred efficiently while maintaining execution order.

Thus, while single-chip runtimes focus on optimizing execution within a single processor, multi-chip runtimes must handle system-wide execution, balancing computation, memory, and interconnect performance.

11.9.2.4 Computation Placement Adaptation

As AI systems expand beyond single-chip execution, computation placement must adapt to account for inter-chip workload distribution and interconnect efficiency. In single-chip accelerators, compilers optimize placement by mapping workloads to tensor cores, vector units, and PEs, ensuring maximum parallelism while minimizing on-chip data movement. However, in multi-chip systems, placement strategies must address interconnect bandwidth constraints, synchronization latency, and hierarchical workload partitioning across multiple accelerators.

Table 11.22 highlights these adaptations. To reduce expensive cross-chip communication, compilers now implement interconnect-aware workload partitioning, strategically assigning computations to accelerators based on communication cost. For instance, in multi-GPU training, compilers optimize placement to minimize NVLink or PCIe traffic, whereas TPU Pods leverage the torus interconnect topology to enhance data exchanges.

Table 11.22: Adaptations in computation placement strategies for multi-chip AI execution.

Aspect	Single-Chip AI Accelerator	Multi-Chip AI System & How Compilers/Runtimes Adapt
Computation Placement	Local PEs, tensor cores, vector units	Hierarchical mapping, interconnect-aware scheduling
Workload Distribution	Optimized within a single chip	Partitioning across accelerators, minimizing inter-chip communication
Synchronization	Managed within local execution units	Runtimes dynamically balance workloads, adjust execution plans

Runtimes complement this by dynamically managing execution workloads, adjusting placement in real-time to balance loads across accelerators. Unlike static compilation, which assumes a fixed hardware topology, AI runtimes continuously monitor system conditions and migrate tasks as needed to prevent bottlenecks. This ensures efficient execution even in environments with fluctuating workload demands or varying hardware availability.

By extending local execution strategies to multi-chip environments, computation placement now requires a careful balance between parallel execution, memory locality, and interconnect-aware scheduling. The next section explores how memory hierarchy must evolve to support efficient execution across distributed AI architectures.

Thus, computation placement at scale builds upon local execution optimizations while introducing new challenges in inter-chip coordination, communication-aware execution, and dynamic load balancing. In the next section, we explore how memory hierarchy must adapt to support efficient execution across multi-chip architectures.

11.9.3 Navigating Multi-Chip AI Complexities

The evolution of AI hardware, from single-chip accelerators to multi-chip systems and wafer-scale integration, highlights the increasing complexity of efficiently executing large-scale machine learning workloads. As we've explored in this chapter, scaling AI systems introduces new challenges in computation placement, memory management, and data movement. While the fundamental principles of AI acceleration remain consistent, their implementation must adapt to the constraints of distributed execution, interconnect bandwidth limitations, and synchronization overhead.

Multi-chip AI architectures represent a significant step forward in addressing the computational demands of modern machine learning models. By distributing workloads across multiple accelerators, these systems offer increased performance, memory capacity, and scalability. However, realizing these benefits requires careful consideration of how computations are mapped to hardware, how memory is partitioned and accessed, and how execution is scheduled across a distributed system.

While we an overview of the key concepts and challenges in multi-chip AI acceleration as they extend beyond a single system, there is still much more to explore. As AI models continue to grow in size and complexity, new architectural innovations, mapping strategies, and runtime optimizations will be needed to sustain efficient execution. The ongoing development of AI hardware and software reflects a broader trend in computing, where specialization and domain-specific architectures are becoming increasingly important for addressing the unique demands of emerging workloads.

Understanding the principles and trade-offs involved in multi-chip AI acceleration enables machine learning engineers and system designers to make informed decisions about how to best deploy and optimize their models. Whether training large language models on TPU pods or deploying computer vision applications on multi-GPU systems, the ability to efficiently map computations to hardware will continue to be a critical factor in realizing the full potential of AI.

11.10 Conclusion

The rapid advancement of machine learning has fundamentally reshaped computer architecture and system design, driving the need for specialized hardware and optimized software to support the increasing computational demands of AI workloads. This chapter has explored the foundational principles of AI acceleration, analyzing how domain-specific architectures, memory hierarchies, and data movement strategies work in concert to maximize performance and mitigate bottlenecks.

We began by examining the historical progression of AI hardware, tracing the shift from general-purpose processors to specialized accelerators tailored for machine learning workloads. This evolution has been driven by the computational intensity of AI models, necessitating vectorized execution, matrix processing, and specialized function units to accelerate key operations.

Memory systems play a pivotal role in AI acceleration, as modern workloads require efficient management of large-scale tensor data across hierarchical memory structures. This chapter detailed the challenges posed by memory bandwidth limitations, irregular access patterns, and off-chip communication, highlighting techniques such as tiling, kernel fusion, and memory-aware data placement that optimize data movement and reuse.

Mapping neural networks to hardware requires balancing computation placement, memory allocation, and execution scheduling. We analyzed key mapping strategies, including weight-stationary, output-stationary, and hybrid approaches, and explored how compilers and runtimes transform high-level models into optimized execution plans that maximize hardware utilization.

As AI workloads scale beyond single-chip accelerators, new challenges emerge in distributed execution, memory coherence, and inter-chip communication. This chapter examined how multi-GPU architectures, TPU pods, and wafer-scale AI systems address these challenges by leveraging hierarchical workload partitioning, distributed memory management, and interconnect-aware scheduling. We also explored how compilers and runtimes must adapt to orchestrate execution across multiple accelerators, ensuring efficient workload distribution and minimizing communication overhead.

The increasing complexity of AI models and the growing scale of machine learning workloads underscore a broader shift in computing—one where specialization and hardware-software co-design are essential for achieving efficiency and scalability. Understanding the fundamental trade-offs in AI acceleration enables system designers, researchers, and engineers to make informed decisions about deploying and optimizing AI models across diverse hardware platforms.

This chapter has provided a comprehensive foundation in AI acceleration, equipping readers with the knowledge to navigate the evolving intersection of machine learning systems, hardware design, and system optimization. As AI continues to advance, the ability to efficiently map computations to hardware will remain a key determinant of performance, scalability, and future innovation in artificial intelligence.

11.11 Resources

ℹ Slides

- *Coming soon.*

❗ Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 12

Benchmarking AI

Figure 12.1: *DALL·E 3 Prompt: Photo of a podium set against a tech-themed backdrop. On each tier of the podium, there are AI chips with intricate designs. The top chip has a gold medal hanging from it, the second one has a silver medal, and the third has a bronze medal. Banners with 'AI Olympics' are displayed prominently in the background.*

Purpose

How can quantitative evaluation reshape the development of machine learning systems, and what metrics reveal true system capabilities?

The measurement and analysis of AI system performance represent a critical element in bridging theoretical capabilities with practical outcomes. Systematic evaluation approaches reveal fundamental relationships between model behavior, resource utilization, and operational reliability. These measurements draw out the essential trade-offs across accuracy, efficiency, and scalability, providing insights that guide architectural decisions throughout the development lifecycle. These evaluation frameworks establish core principles for assessing and validating system design choices and enable the creation of robust solutions that meet increasingly complex performance requirements across diverse deployment scenarios.

Learning Objectives

- Understand the objectives of AI benchmarking, including performance evaluation, resource assessment, and validation.
- Differentiate between training and inference benchmarking and their respective evaluation methodologies.
- Identify key benchmarking metrics and trends, including accuracy, fairness, complexity, and efficiency.
- Recognize system benchmarking concepts, including throughput, latency, power consumption, and computational efficiency.
- Understand the limitations of isolated evaluations and the necessity of integrated benchmarking frameworks.

12.1 Overview

Computing systems continue to evolve and grow in complexity. Understanding their performance becomes essential to engineer them better. System evaluation measures how computing systems perform relative to specified requirements and goals. Engineers and researchers examine metrics like processing speed, resource usage, and reliability to understand system behavior under different conditions and workloads. These measurements help teams identify bottlenecks, optimize performance, and verify that systems meet design specifications.

Standardized measurement forms the backbone of scientific and engineering progress. The metric system enables precise communication of physical quantities. Organizations like the National Institute of Standards and Technology maintain fundamental measures from the kilogram to the second. This standardization extends to computing, where benchmarks provide uniform methods to quantify system performance. Standard performance tests measure processor operations, memory bandwidth, network throughput, and other computing capabilities. These benchmarks allow meaningful comparison between different hardware and software configurations.

Machine learning systems present distinct measurement challenges. Unlike traditional computing tasks, ML systems integrate hardware performance, algorithmic behavior, and data characteristics. Performance evaluation must account for computational efficiency and statistical effectiveness. Training time, model accuracy, and generalization capabilities all factor into system assessment. The interdependence between computing resources, algorithmic choices, and dataset properties creates new dimensions for measurement and comparison.

These considerations lead us to define machine learning benchmarking as follows:

ℹ Definition of ML Benchmarking

Machine Learning Benchmarking (ML Benchmarking) is the *systematic evaluation* of *compute performance, algorithmic effectiveness, and data quality* in machine learning systems. It assesses *system capabilities*, *model accuracy and convergence*, and *data scalability and representativeness* to optimize system performance across diverse workloads. ML benchmarking enables engineers and researchers to *quantify trade-offs*, *improve deployment efficiency*, and *ensure reproducibility* in both research and production settings. As ML systems evolve, benchmarks also incorporate *fairness, robustness, and energy efficiency*, reflecting the increasing complexity of AI evaluation.

This chapter focuses primarily on benchmarking machine learning systems, examining how computational resources affect training and inference performance. While the main emphasis remains on system-level evaluation, understanding the role of algorithms and data proves essential for comprehensive ML benchmarking.

12.2 Historical Context

The evolution of computing benchmarks mirrors the development of computer systems themselves, progressing from simple performance metrics to increasingly specialized evaluation frameworks. As computing expanded beyond scientific calculations into diverse applications, benchmarks evolved to measure new capabilities, constraints, and use cases. This progression reflects three major shifts in computing: the transition from mainframes to personal computers, the rise of energy efficiency as a critical concern, and the emergence of specialized computing domains such as machine learning.

Early benchmarks focused primarily on raw computational power, measuring basic operations like floating-point calculations. As computing applications diversified, benchmark development branched into distinct specialized categories, each designed to evaluate specific aspects of system performance. This specialization accelerated with the emergence of graphics processing, mobile computing, and eventually, cloud services and machine learning.

12.2.1 Performance Benchmarks

The evolution of benchmarks in computing illustrates how systematic performance measurement has shaped technological progress. During the 1960s and 1970s, when mainframe computers dominated the computing landscape, performance benchmarks focused primarily on fundamental computational tasks. The Whetstone benchmark[0], introduced in 1964 to measure floating-point arithmetic performance, became a definitive standard that demonstrated how systematic testing could drive improvements in computer architecture (Curnow 1976).

The introduction of the LINPACK benchmark in 1979 expanded the focus of performance evaluation, offering a means to assess how efficiently systems

[0] Introduced in 1964, the Whetstone benchmark was one of the first synthetic benchmarks designed to measure floating-point arithmetic performance, influencing early computer architecture improvements.

solved linear equations. As computing shifted toward personal computers in the 1980s, the need for standardized performance measurement grew. The Dhrystone benchmark, introduced in 1984, provided one of the first integer-based benchmarks, complementing floating-point evaluations (Weicker 1984).

The late 1980s and early 1990s saw the emergence of systematic benchmarking frameworks that emphasized real-world workloads. The SPEC CPU benchmarks[1], introduced in 1989 by the System Performance Evaluation Cooperative (SPEC), fundamentally changed hardware evaluation by shifting the focus from synthetic tests to a standardized suite designed to measure performance using practical computing workloads. This approach enabled manufacturers to optimize their systems for real applications, accelerating advances in processor design and software optimization.

The increasing demand for graphics-intensive applications and mobile computing in the 1990s and early 2000s presented new benchmarking challenges. The introduction of 3DMark in 1998 established an industry standard for evaluating graphics performance, shaping the development of programmable shaders and modern GPU architectures. Mobile computing introduced an additional constraint, namely, power efficiency, necessitating benchmarks that assessed both computational performance and energy consumption. The release of MobileMark by BAPCo provided a means to evaluate power efficiency in laptops and mobile devices, influencing the development of energy-efficient architectures such as ARM.

The focus of benchmarking in the past decade has shifted toward cloud computing, big data, and artificial intelligence. Cloud service providers such as Amazon Web Services and Google Cloud optimize their platforms based on performance, scalability, and cost-effectiveness (Ranganathan and Hölzle 2024). Benchmarks like CloudSuite have become critical for evaluating cloud infrastructure, measuring how well systems handle distributed workloads. Machine learning has introduced another dimension of performance evaluation. The introduction of MLPerf in 2018 established a widely accepted standard for measuring machine learning training and inference efficiency across different hardware architectures.

12.2.2 Energy Benchmarks

As computing scaled from personal devices to massive data centers, energy efficiency emerged as a critical dimension of performance evaluation. The mid-2000s marked a shift in benchmarking methodologies, moving beyond raw computational speed to assess power efficiency across diverse computing platforms. The increasing thermal constraints in processor design, coupled with the scaling demands of large-scale internet services, underscored energy consumption as a fundamental consideration in system evaluation (Barroso and Hölzle 2007b).

Power benchmarking addresses three interconnected challenges: environmental sustainability, operational efficiency, and device usability. The growing energy demands of the technology sector have intensified concerns about sustainability, while energy costs continue to shape the economics of data center operations. In mobile computing, power efficiency directly determines battery

[1] Launched in 1989, the SPEC CPU benchmark suite shifted performance evaluation towards real-world workloads, significantly influencing processor design and optimization.

life and user experience, reinforcing the importance of energy-aware performance measurement.

The industry has responded with several standardized benchmarks that quantify energy efficiency. SPEC Power provides a widely accepted methodology for measuring server efficiency across varying workload levels, allowing for direct comparisons of power-performance trade-offs. The Green500 ranking[2] applies similar principles to high-performance computing, ranking the world's most powerful supercomputers based on their energy efficiency rather than their raw performance. The ENERGY STAR certification program has also established foundational energy standards that have shaped the design of consumer and enterprise computing systems.

Power benchmarking faces distinct challenges, particularly in accounting for the diverse workload patterns and system configurations encountered across different computing environments. Recent advancements, such as the MLPerf Power benchmark, have introduced specialized methodologies for measuring the energy impact of machine learning workloads, addressing the growing importance of energy efficiency in AI-driven computing. As artificial intelligence and edge computing continue to evolve, power benchmarking will play an increasingly crucial role in driving energy-efficient hardware and software innovations.

12.2.3 Domain-Specific Benchmarks

The evolution of computing applications, particularly in artificial intelligence, has highlighted the limitations of general-purpose benchmarks and led to the development of domain-specific evaluation frameworks. Standardized benchmarks, while effective for assessing broad system performance, often fail to capture the unique constraints and operational requirements of specialized workloads. This gap has resulted in the emergence of tailored benchmarking methodologies designed to evaluate performance in specific computing domains (John L. Hennessy and Patterson 2003).

Machine learning presents one of the most prominent examples of this transition. Traditional CPU and GPU benchmarks are insufficient for assessing workloads, which involve complex interactions between computation, memory bandwidth, and data movement. The introduction of MLPerf has standardized performance measurement for machine learning models, providing detailed insights into training and inference efficiency.

Beyond AI, domain-specific benchmarks have been adopted across various industries. Healthcare organizations have developed benchmarking frameworks to evaluate machine learning models used in medical diagnostics, ensuring that performance assessments align with real-world patient data. In financial computing, specialized benchmarking methodologies assess transaction latency and fraud detection accuracy, ensuring that high-frequency trading systems meet stringent timing requirements. Autonomous vehicle developers implement evaluation frameworks that test AI models under varying environmental conditions and traffic scenarios, ensuring the reliability of self-driving systems.

The strength of domain-specific benchmarks lies in their ability to capture workload-specific performance characteristics that general benchmarks may

[2] Established in 2007, the Green500 ranks supercomputers based on energy efficiency, highlighting advances in power-efficient high-performance computing.

overlook. By tailoring performance evaluation to sector-specific requirements, these benchmarks provide insights that drive targeted optimizations in both hardware and software. As computing continues to expand into new domains, specialized benchmarking will remain a key tool for assessing and improving performance in emerging fields.

12.3 AI Benchmarks

The evolution of benchmarks reaches its apex in machine learning, reflecting a journey that parallels the field's development towards domain-specific applications. Early machine learning benchmarks focused primarily on algorithmic performance, measuring how well models could perform specific tasks (Lecun et al. 1998). As machine learning applications scaled and computational demands grew, the focus expanded to include system performance and hardware efficiency (Jouppi, Young, et al. 2017a). Most recently, the critical role of data quality has emerged as the third essential dimension of evaluation (Gebru et al. 2021b).

What sets AI benchmarks apart from traditional performance metrics is their inherent variability, introducing accuracy as a fundamental dimension of evaluation. Unlike conventional benchmarks, which measure fixed, deterministic characteristics like computational speed or energy consumption, AI benchmarks must account for the probabilistic nature of machine learning models. The same system can produce different results depending on the data it encounters, making accuracy a defining factor in performance assessment. This distinction adds complexity, as benchmarking AI systems requires not only measuring raw computational efficiency but also understanding trade-offs between accuracy, generalization, and resource constraints.

The growing complexity and ubiquity of machine learning systems demand comprehensive benchmarking across all three dimensions: algorithmic models, hardware systems, and training data. This multifaceted evaluation approach represents a significant departure from earlier benchmarks that could focus on isolated aspects like computational speed or energy efficiency (Hernandez and Brown 2020). Modern machine learning benchmarks must address the sophisticated interplay between these dimensions, as limitations in any one area can fundamentally constrain overall system performance.

This evolution in benchmark complexity mirrors the field's deepening understanding of what drives machine learning system success. While algorithmic innovations initially dominated progress metrics, the challenges of deploying models at scale revealed the critical importance of hardware efficiency (Jouppi et al. 2021b). Subsequently, high-profile failures of machine learning systems in real-world deployments highlighted how data quality and representation fundamentally determine system reliability and fairness (Bender et al. 2021). Understanding how these dimensions interact has become essential for accurately assessing machine learning system performance, informing development decisions, and measuring technological progress in the field.

12.3.1 Algorithmic Benchmarks

AI algorithms must balance multiple interconnected performance objectives, including accuracy, speed, resource efficiency, and generalization capability. As machine learning applications span diverse domains, including computer vision, natural language processing, speech recognition, and reinforcement learning, evaluating these objectives requires standardized methodologies tailored to each domain's unique challenges. Algorithmic benchmarks, such as ImageNet (J. Deng et al. 2009), establish these evaluation frameworks, providing a consistent basis for comparing different machine learning approaches.

ℹ Definition of Machine Learning Algorithmic Benchmarks

ML Algorithmic benchmarks refer to the evaluation of machine learning models on *standardized tasks* using *predefined datasets and metrics*. These benchmarks measure *accuracy, efficiency, and generalization* to ensure *objective comparisons* across different models. Algorithmic benchmarks provide *performance baselines*, enabling systematic assessment of *trade-offs between model complexity and computational cost*. They drive *technological progress* by tracking improvements over time and identifying *limitations* in existing approaches.

Algorithmic benchmarks serve several critical functions in advancing AI. They establish clear performance baselines, enabling objective comparisons between competing approaches. By systematically evaluating trade-offs between model complexity, computational requirements, and task performance, they help researchers and practitioners identify optimal design choices. Moreover, they track technological progress by documenting improvements over time, guiding the development of new techniques while exposing limitations in existing methodologies.

For instance, the graph in Figure 12.2 illustrates the significant reduction in error rates on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) classification task over the years. Starting from the baseline models in 2010 and 2011, the introduction of AlexNet in 2012 marked a substantial improvement, reducing the error rate from 25.8% to 16.4%. Subsequent models like ZFNet, VGGNet, GoogleNet, and ResNet continued this trend, with ResNet achieving a remarkable error rate of 3.57% by 2015. This progression highlights how algorithmic benchmarks not only measure current capabilities but also drive continuous advancements in AI performance.

12.3.2 System Benchmarks

AI computations, particularly in machine learning, place extraordinary demands on computational resources. The underlying hardware infrastructure, encompassing general-purpose CPUs, graphics processing units (GPUs), tensor processing units (TPUs), and application-specific integrated circuits (ASICs), fundamentally determines the speed, efficiency, and scalability of AI solutions. System benchmarks establish standardized methodologies for evaluating hardware performance across diverse AI workloads, measuring critical metrics

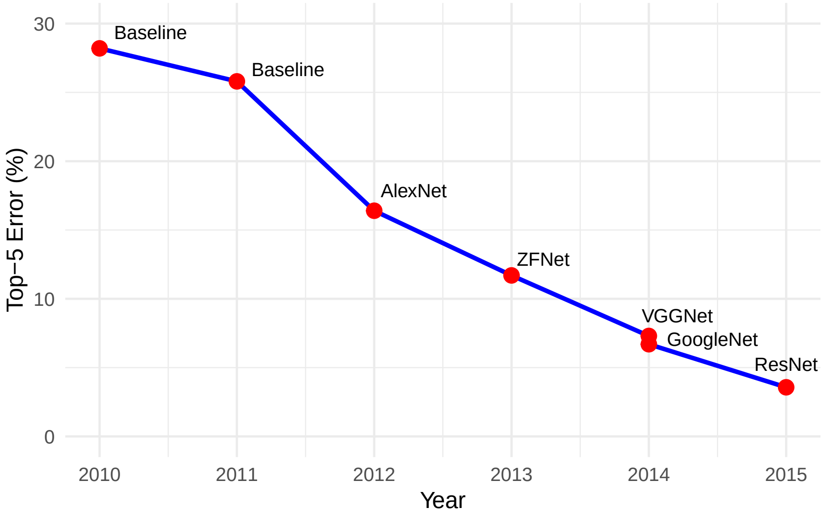


Figure 12.2: ImageNet accuracy improvements over the years.

including computational throughput, memory bandwidth, power efficiency, and scaling characteristics (Reddi et al. 2019; Mattson et al. 2020).

ℹ Definition of Machine Learning System Benchmarks

ML System benchmarks refer to the evaluation of *computational infrastructure* used to execute AI workloads, assessing *performance, efficiency, and scalability* under standardized conditions. These benchmarks measure *throughput, latency, and resource utilization* to ensure *objective comparisons* across different system configurations. System benchmarks provide *insights into workload efficiency*, guiding *infrastructure selection, system optimization*, and *advancements in computational architectures*.

These benchmarks fulfill two essential functions in the AI ecosystem. First, they enable developers and organizations to make informed decisions when selecting hardware platforms for their AI applications by providing comprehensive comparative performance data across system configurations. Critical evaluation factors include training speed, inference latency, energy efficiency, and cost-effectiveness. Second, hardware manufacturers rely on these benchmarks to quantify generational improvements and guide the development of specialized AI accelerators, driving continuous advancement in computational capabilities.

System benchmarks evaluate performance across multiple scales, ranging from single-chip configurations to large distributed systems, and diverse AI workloads including both training and inference tasks. This comprehensive evaluation approach ensures that benchmarks accurately reflect real-world deployment scenarios and deliver actionable insights that inform both hardware selection decisions and system architecture design. For example, Figure 12.3

illustrates the correlation between ImageNet classification error rates and GPU adoption from 2010 to 2014. These results clearly highlight how improved hardware capabilities, combined with algorithmic advances, drove significant progress in computer vision performance.

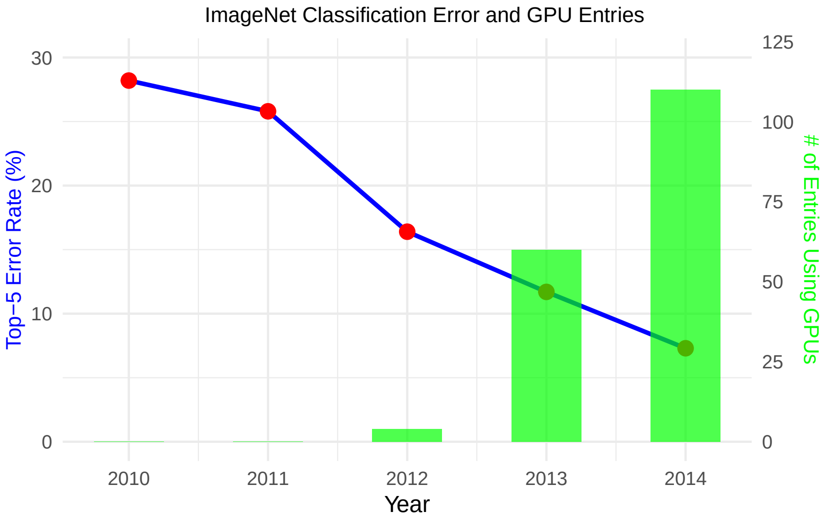


Figure 12.3: ImageNet accuracy improvements and use of GPUs since the dawn of AlexNet in 2012.

12.3.3 Data Benchmarks

Data quality, scale, and diversity fundamentally shape machine learning system performance, directly influencing how effectively algorithms learn and generalize to new situations. Data benchmarks establish standardized datasets and evaluation methodologies that enable consistent comparison of different approaches. These frameworks assess critical aspects of data quality, including domain coverage, potential biases, and resilience to real-world variations in input data (Gebru et al. 2021b).

i Definition of Machine Learning Data Benchmarks

ML Data benchmarks refer to the evaluation of *datasets and data quality* in machine learning, assessing *coverage, bias, and robustness* under standardized conditions. These benchmarks measure *data representativeness, consistency, and impact on model performance* to ensure *objective comparisons* across different AI approaches. Data benchmarks provide *insights into data reliability*, guiding *dataset selection, bias mitigation*, and *improvements in data-driven AI systems*.

Data benchmarks serve an essential function in understanding AI system behavior under diverse data conditions. Through systematic evaluation, they help identify common failure modes, expose gaps in data coverage, and reveal underlying biases that could impact model behavior in deployment. By providing common frameworks for data evaluation, these benchmarks enable the AI community to systematically improve data quality and address potential issues before deploying systems in production environments. This proactive approach to data quality assessment has become increasingly critical as AI systems take on more complex and consequential tasks across different domains.

12.3.4 Community Consensus

The proliferation of benchmarks spanning performance, energy efficiency, and domain-specific applications creates a fundamental challenge: establishing industry-wide standards. While early computing benchmarks primarily measured processor speed and memory bandwidth, modern benchmarks evaluate sophisticated aspects of system performance, from power consumption profiles to application-specific capabilities. This evolution in scope and complexity necessitates comprehensive validation and consensus from the computing community, particularly in rapidly evolving fields like machine learning where performance must be evaluated across multiple interdependent dimensions.

The lasting impact of a benchmark depends fundamentally on its acceptance by the research community, where technical excellence alone proves insufficient. Benchmarks developed without broad community input often fail to gain traction, frequently missing metrics that leading research groups consider essential. Successful benchmarks emerge through collaborative development involving academic institutions, industry partners, and domain experts. This inclusive approach ensures benchmarks evaluate capabilities most crucial for advancing the field, while balancing theoretical and practical considerations.

Benchmarks developed through extensive collaboration among respected institutions carry the authority necessary to drive widespread adoption, while those perceived as advancing particular corporate interests face skepticism and limited acceptance. The success of ImageNet demonstrates how sustained community engagement through workshops and challenges establishes long-term viability. This community-driven development creates a foundation for formal standardization, where organizations like IEEE and ISO transform these benchmarks into official standards.

The standardization process provides crucial infrastructure for benchmark formalization and adoption. IEEE working groups transform community-developed benchmarking methodologies into formal industry standards, establishing precise specifications for measurement and reporting. The IEEE 2416-2019 standard for system power modeling[3] exemplifies this process, codifying best practices developed through community consensus. Similarly, ISO/IEC technical committees develop international standards for benchmark validation and certification, ensuring consistent evaluation across global research and industry communities. These organizations bridge the gap between community-driven innovation and formal standardization, providing frameworks that enable reliable comparison of results across different institutions and geographic regions.

Successful community benchmarks establish clear governance structures for managing their evolution. Through rigorous version control systems and

[3] IEEE 2416-2019: A standard defining methodologies for parameterized power modeling, enabling system-level power analysis and optimization in electronic design, including AI hardware.

detailed change documentation, benchmarks maintain backward compatibility while incorporating new advances. This governance includes formal processes for proposing, reviewing, and implementing changes, ensuring that benchmarks remain relevant while maintaining stability. Modern benchmarks increasingly emphasize reproducibility requirements, incorporating automated verification systems and standardized evaluation environments.

Open access accelerates benchmark adoption and ensures consistent implementation. Projects that provide open-source reference implementations, comprehensive documentation, validation suites, and containerized evaluation environments reduce barriers to entry. This standardization enables research groups to evaluate solutions using uniform methods and metrics. Without such coordinated implementation frameworks, organizations might interpret benchmarks inconsistently, compromising result reproducibility and meaningful comparison across studies.

The most successful benchmarks strike a careful balance between academic rigor and industry practicality. Academic involvement ensures theoretical soundness and comprehensive evaluation methodology, while industry participation grounds benchmarks in practical constraints and real-world applications. This balance proves particularly crucial in machine learning benchmarks, where theoretical advances must translate to practical improvements in deployed systems (D. Patterson et al. 2021b).

Community consensus establishes enduring benchmark relevance, while fragmentation impedes scientific progress. Through collaborative development and transparent operation, benchmarks evolve into authoritative standards for measuring advancement. The most successful benchmarks in energy efficiency and domain-specific applications share this foundation of community development and governance, demonstrating how collective expertise and shared purpose create lasting impact in rapidly advancing fields.

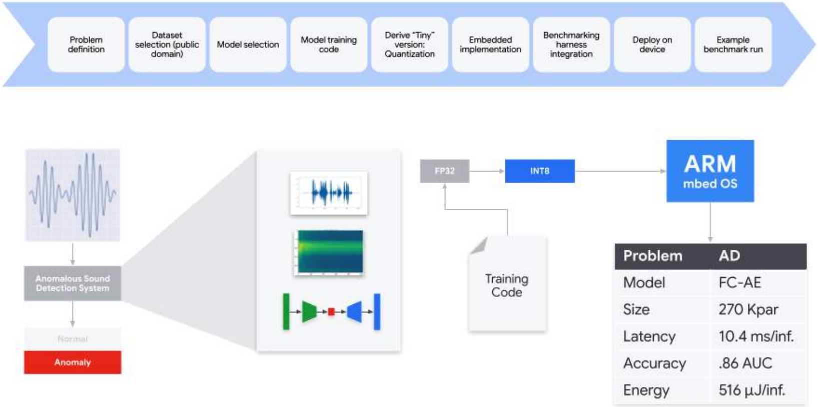
12.4 Benchmark Components

An AI benchmark provides a structured framework for evaluating artificial intelligence systems. While individual benchmarks vary in their specific focus, they share common components that enable systematic evaluation and comparison of AI models.

Figure 12.4 illustrates the structured workflow of a benchmark implementation, showcasing how components like task definition, dataset selection, model selection, and evaluation interconnect to form a complete evaluation pipeline. This visualization highlights how each phase builds upon the previous one, ensuring systematic and reproducible AI performance assessment.

12.4.1 Problem Definition

A benchmark implementation begins with a formal specification of the machine learning task and its evaluation criteria. In machine learning, tasks represent well-defined problems that AI systems must solve. Consider an anomaly detection system that processes audio signals to identify deviations from normal



Problem	AD
Model	FC-AE
Size	270 Kpar
Latency	10.4 ms/inf.
Accuracy	.86 AUC
Energy	516 µJ/inf.

Figure 12.4: Example of benchmark components.

operation patterns, as shown in Figure 12.4. This industrial monitoring application exemplifies how formal task specifications translate into practical implementations.

The formal definition of a benchmark task encompasses both the computational problem and its evaluation framework. While the specific tasks vary by domain, well-established categories have emerged across fields. Natural language processing tasks, for example, include machine translation, question answering (Hirschberg and Manning 2015), and text classification. Computer vision similarly employs standardized tasks such as object detection, image segmentation, and facial recognition (Everingham et al. 2009).

Every benchmark task specification must define three fundamental elements. The input specification determines what data the system processes. In Figure 12.4, this consists of audio waveform data. The output specification describes the required system response, such as the binary classification of normal versus anomalous patterns. The performance specification establishes quantitative requirements for accuracy, processing speed, and resource utilization.

Task design directly impacts the benchmark's ability to evaluate AI systems effectively. The audio anomaly detection example illustrates this relationship through its specific requirements: processing continuous signal data, adapting to varying noise conditions, and operating within strict time constraints. These practical constraints create a detailed framework for assessing model performance, ensuring evaluations reflect real-world operational demands.

The implementation of a benchmark proceeds systematically from this task definition. Each subsequent phase - from dataset selection through deployment - builds upon these initial specifications, ensuring that evaluations maintain consistency while addressing the defined requirements across different approaches and implementations.

12.4.2 Standardized Datasets

Building upon the problem definition, standardized datasets provide the foundation for training and evaluating models. These carefully curated collections ensure all models undergo testing under identical conditions, enabling direct comparisons across different approaches and architectures. Figure 12.4 demonstrates this through an audio anomaly detection example, where waveform data serves as the standardized input for evaluating detection performance.

In computer vision, datasets such as ImageNet (J. Deng et al. 2009), COCO (T.-Y. Lin et al. 2014), and CIFAR-10 (Krizhevsky, Hinton, et al. 2009) serve as reference standards. For natural language processing, collections such as SQuAD (Rajpurkar et al. 2016), GLUE (A. Wang et al. 2018), and WikiText (Merity et al. 2016) fulfill similar functions. These datasets encompass a range of complexities and edge cases to thoroughly evaluate machine learning systems.

The strategic selection of datasets, shown early in the workflow of Figure 12.4, shapes all subsequent implementation steps and determines the benchmark's effectiveness. In the audio anomaly detection example, the dataset must include representative waveform samples of normal operation alongside examples of various anomalous conditions. Notable examples include datasets like ToyADMOS for industrial manufacturing anomalies and Google Speech Commands for general sound recognition. Regardless of the specific dataset chosen, the data volume must suffice for both model training and validation, while incorporating real-world signal characteristics and noise patterns that reflect deployment conditions.

The selection of benchmark datasets fundamentally shapes experimental outcomes and model evaluation. Effective datasets must balance two key requirements: accurately representing real-world challenges while maintaining sufficient complexity to differentiate model performance meaningfully. While research often utilizes simplified datasets like ToyADMOS (Koizumi et al. 2019), these controlled environments, though valuable for methodological development, may not fully capture real-world deployment complexities. Benchmark development frequently necessitates combining multiple datasets due to access limitations on proprietary industrial data. As machine learning capabilities advance, benchmark datasets must similarly evolve to maintain their utility in evaluating contemporary systems and emerging challenges.

12.4.3 Model Selection

The benchmark process advances systematically from initial task definition to model architecture selection and implementation. This critical phase establishes performance baselines and determines the optimal modeling approach. Figure 12.4 illustrates this progression through the model selection stage and subsequent training code development.

Baseline models serve as the reference points for evaluating novel approaches. These span from basic implementations, including linear regression for continuous predictions and logistic regression for classification tasks, to advanced architectures with proven success in comparable domains. In natural language processing applications, transformer-based models like BERT have emerged as standard benchmarks for comparative analysis.

Selecting the right baseline model requires careful evaluation of architectures against benchmark requirements. This selection process directly informs the development of training code, which forms the cornerstone of benchmark reproducibility. The training implementation must thoroughly document all aspects of the model pipeline, from data preprocessing through training procedures, enabling precise replication of model behavior across research teams.

Model development follows two primary optimization paths: training and inference. During training optimization, efforts concentrate on achieving target accuracy metrics while operating within computational constraints. The training implementation must demonstrate consistent achievement of performance thresholds under specified conditions.

The inference optimization path addresses deployment considerations, particularly the transition from development to production environments. A key example involves precision reduction through quantization, progressing from FP32 to INT8 representations to enhance deployment efficiency. This process demands careful calibration to maintain model accuracy while reducing resource requirements. The benchmark must detail both the quantization methodology and verification procedures that confirm preserved performance.

The intersection of these optimization paths with real-world constraints shapes deployment strategy. Comprehensive benchmarks must therefore specify requirements for both training and inference scenarios, ensuring models maintain consistent performance from development through deployment. This crucial connection between development and production metrics naturally leads to the establishment of evaluation criteria.

The optimization process must balance four key objectives: model accuracy, computational speed, memory utilization, and energy efficiency. This complex optimization landscape necessitates robust evaluation metrics that can effectively quantify performance across all dimensions. As models transition from development to deployment, these metrics serve as critical tools for guiding optimization decisions and validating performance enhancements.

12.4.4 Evaluation Metrics

While model selection establishes the architectural framework, evaluation metrics provide the quantitative measures needed to assess machine learning model performance. These metrics establish objective standards for comparing different approaches, enabling researchers and practitioners to gauge solution effectiveness. The selection of appropriate metrics represents a fundamental aspect of benchmark design, as they must align with task objectives while providing meaningful insights into model behavior across both training and deployment scenarios.

Task-specific metrics quantify a model's performance on its intended function. Classification tasks employ metrics including accuracy (overall correct predictions), precision (positive prediction accuracy), recall (positive case detection rate), and F1 score (precision-recall harmonic mean) (Sokolova and Lapalme 2009). Regression problems utilize error measurements like Mean Squared Error (MSE) and Mean Absolute Error (MAE) to assess prediction accuracy. Domain-specific applications often require specialized metrics - for example, machine translation uses the BLEU score to evaluate the semantic and syntactic similarity between machine-generated and human reference translations (Papineni et al. 2001).

As models transition from research to production deployment, implementation metrics become equally important. Model size, measured in parameters or memory footprint, affects deployment feasibility across different hardware platforms. Processing latency, typically measured in milliseconds per inference, determines whether the model meets real-time requirements. Energy consumption, measured in watts or joules per inference, indicates operational efficiency. These practical considerations reflect the growing need for solutions that balance accuracy with computational efficiency.

The selection of appropriate metrics requires careful consideration of task requirements and deployment constraints. A single metric rarely captures all relevant aspects of performance. For instance, in anomaly detection systems, high accuracy alone may not indicate good performance if the model generates frequent false alarms. Similarly, a fast model with poor accuracy fails to provide practical value.

Figure 12.4 demonstrates this multi-metric evaluation approach. The anomaly detection system reports performance across multiple dimensions: model size (270 Kparameters), processing speed (10.4 ms/inference), and detection accuracy (0.86 AUC). This combination of metrics ensures the model meets both technical and operational requirements in real-world deployment scenarios.

12.4.5 Benchmark Harness

Evaluation metrics provide the measurement framework, while a benchmark harness implements the systematic infrastructure for evaluating model performance under controlled conditions. This critical component ensures reproducible testing by managing how inputs are delivered to the system under test and how measurements are collected, effectively transforming theoretical metrics into quantifiable measurements.

The harness design should align with the intended deployment scenario and usage patterns. For server deployments, the harness implements request patterns that simulate real-world traffic, typically generating inputs using a Poisson distribution to model random but statistically consistent server workloads. The harness manages concurrent requests and varying load intensities to evaluate system behavior under different operational conditions.

For embedded and mobile applications, the harness generates input patterns that reflect actual deployment conditions. This might involve sequential image injection for mobile vision applications or synchronized multi-sensor streams for autonomous systems. Such precise input generation and timing control ensures the system experiences realistic operational patterns, revealing performance characteristics that would emerge in actual device deployment.

The harness must also accommodate different throughput models. Batch processing scenarios require the ability to evaluate system performance on large volumes of parallel inputs, while real-time applications need precise timing control for sequential processing. Figure 12.4 illustrates this in the embedded implementation phase, where the harness must support precise measurement of inference time and energy consumption per operation.

Reproducibility demands that the harness maintain consistent testing conditions across different evaluation runs. This includes controlling environmental factors such as background processes, thermal conditions, and power states that might affect performance measurements. The harness must also provide mechanisms for collecting and logging performance metrics without significantly impacting the system under test.

12.4.6 System Specifications

Beyond the benchmark harness that controls test execution, system specifications are fundamental components of machine learning benchmarks that directly impact model performance, training time, and experimental reproducibility. These specifications encompass the complete computational environment, ensuring that benchmarking results can be properly contextualized, compared, and reproduced by other researchers.

Hardware specifications typically include:

1. Processor type and speed (e.g., CPU model, clock rate)
2. GPUs, or TPUs, including model, memory capacity, and quantity if used for distributed training
3. Memory capacity and type (e.g., RAM size, DDR4)
4. Storage type and capacity (e.g., SSD, HDD)
5. Network configuration, if relevant for distributed computing

Software specifications generally include:

1. Operating system and version
2. Programming language and version
3. Machine learning frameworks and libraries (e.g., TensorFlow, PyTorch) with version numbers
4. Compiler information and optimization flags
5. Custom software or scripts used in the benchmark process
6. Environment management tools and configuration (e.g., Docker containers, virtual environments)

The precise documentation of these specifications is essential for experimental validity and reproducibility. This documentation enables other researchers to replicate the benchmark environment with high fidelity, provides critical context for interpreting performance metrics, and facilitates understanding of resource requirements and scaling characteristics across different models and tasks.

In many cases, benchmarks may include results from multiple hardware configurations to provide a more comprehensive view of model performance across different computational environments. This approach is particularly valuable as it highlights the trade-offs between model complexity, computational resources, and performance.

As the field evolves, hardware and software specifications increasingly incorporate detailed energy consumption metrics and computational efficiency measures, such as FLOPS/watt and total power usage over training time. This expansion reflects growing concerns about the environmental impact of large-scale machine learning models and supports the development of more sustainable AI practices. Comprehensive specification documentation thus serves multiple purposes: enabling reproducibility, supporting fair comparisons, and advancing both the technical and environmental aspects of machine learning research.

12.4.7 Run Rules

Run rules establish the procedural framework that ensures benchmark results can be reliably replicated by researchers and practitioners, complementing the technical environment defined by system specifications. These guidelines are fundamental for validating research claims, building upon existing work, and advancing machine learning. Central to reproducibility in AI benchmarks is the management of controlled randomness—the systematic handling of stochastic processes such as weight initialization and data shuffling that ensures consistent, verifiable results.

Comprehensive documentation of hyperparameters forms a critical component of reproducibility. Hyperparameters are configuration settings that govern the learning process independently of the training data, including learning rates, batch sizes, and network architectures. Given that minor hyperparameter adjustments can significantly impact model performance, their precise documentation is essential. Additionally, benchmarks mandate the preservation and sharing of training and evaluation datasets. When direct data sharing is restricted by privacy or licensing constraints, benchmarks must provide detailed specifications for data preprocessing and selection criteria, enabling researchers to construct comparable datasets or understand the characteristics of the original experimental data.

Code provenance and availability constitute another vital aspect of reproducibility guidelines. Contemporary benchmarks typically require researchers to publish implementation code in version-controlled repositories, encompassing not only the model implementation but also comprehensive scripts for data preprocessing, training, and evaluation. Advanced benchmarks often provide containerized environments that encapsulate all dependencies and configurations. Furthermore, detailed experimental logging is mandatory, including systematic recording of training metrics, model checkpoints, and documentation of any experimental adjustments.

These reproducibility guidelines serve multiple crucial functions: they enhance transparency, enable rigorous peer review, and accelerate scientific progress in AI research. By following these protocols, the research community can effectively verify results, iterate on successful approaches, and identify methodological limitations. In the rapidly evolving landscape of machine learning, these robust reproducibility practices form the foundation for reliable and progressive research.

12.4.8 Result Interpretation

Building upon the foundation established by run rules, result interpretation guidelines provide the essential framework for understanding and contextualizing benchmark outcomes. These guidelines help researchers and practitioners draw meaningful conclusions from benchmark results, ensuring fair and informative comparisons between different models or approaches. A fundamental aspect is understanding the statistical significance of performance differences. Benchmarks typically specify protocols for conducting statistical tests and reporting confidence intervals, enabling practitioners to distinguish between meaningful improvements and variations attributable to random factors.

Result interpretation requires careful consideration of real-world applications. While a 1% improvement in accuracy might be crucial for medical diagnostics or financial systems, other applications might prioritize inference speed or model efficiency over marginal accuracy gains. Understanding these context-specific requirements is essential for meaningful interpretation of benchmark results. Users must also recognize inherent benchmark limitations, as no single evaluation framework can encompass all possible use cases. Common limitations include dataset biases, task-specific characteristics, and constraints of evaluation metrics.

Modern benchmarks often necessitate multi-dimensional analysis across various performance metrics. For instance, when a model demonstrates superior accuracy but requires substantially more computational resources, interpretation guidelines help practitioners evaluate these trade-offs based on their specific constraints and requirements. The guidelines also address the critical issue of benchmark overfitting, where models might be excessively optimized for specific benchmark tasks at the expense of real-world generalization. To mitigate this risk, guidelines often recommend evaluating model performance on related but distinct tasks and considering practical deployment scenarios.

These comprehensive interpretation frameworks ensure that benchmarks serve their intended purpose: providing standardized performance measurements while enabling nuanced understanding of model capabilities. This balanced approach supports evidence-based decision-making in both research contexts and practical machine learning applications.

12.4.9 Example Benchmark

A benchmark run evaluates system performance by synthesizing multiple components under controlled conditions to produce reproducible measurements. Figure 12.4 illustrates this integration through an audio anomaly detection system. It shows how performance metrics are systematically measured and reported within a framework that encompasses problem definition, datasets, model selection, evaluation criteria, and standardized run rules.

The benchmark measures several key performance dimensions. For computational resources, the system reports a model size of 270 Kparameters and requires 10.4 milliseconds per inference. For task effectiveness, it achieves a detection accuracy of 0.86 AUC (Area Under Curve) in distinguishing normal from anomalous audio patterns. For operational efficiency, it consumes 516 µJ of energy per inference.

The relative importance of these metrics varies by deployment context. Energy consumption per inference is critical for battery-powered devices but less consequential for systems with constant power supply. Model size constraints differ significantly between cloud deployments with abundant resources and embedded devices with limited memory. Processing speed requirements depend on whether the system must operate in real-time or can process data in batches.

The benchmark reveals inherent trade-offs between performance metrics in machine learning systems. For instance, reducing the model size from 270 Kparameters might improve processing speed and energy efficiency but could decrease the 0.86 AUC detection accuracy. Figure 12.4 illustrates how these interconnected metrics contribute to overall system performance in the deployment phase.

Whether these measurements constitute a "passing" benchmark depends on the specific requirements of the intended application. The benchmark framework provides the structure and methodology for consistent evaluation, while the acceptance criteria must align with deployment constraints and performance requirements.

12.5 Benchmarking Granularity

While benchmarking components individually provides detailed insights into model selection, dataset efficiency, and evaluation metrics, a complete assessment of machine learning systems requires analyzing performance across different levels of abstraction. Benchmarks can range from fine-grained evaluations of individual tensor operations to holistic end-to-end measurements of full AI pipelines.

System level benchmarking provides a structured and systematic approach to assessing a ML system's performance across various dimensions. Given the complexity of ML systems, we can dissect their performance through different levels of granularity and obtain a comprehensive view of the system's efficiency, identify potential bottlenecks, and pinpoint areas for improvement. To this end, various types of benchmarks have evolved over the years and continue to persist.

Figure 12.5 shows the different layers of granularity of an ML system. At the application level, end-to-end benchmarks assess the overall system performance, considering factors like data preprocessing, model training, and inference. While at the model layer, benchmarks focus on assessing the efficiency and accuracy of specific models. This includes evaluating how well models generalize to new data and their computational efficiency during training and inference. Furthermore, benchmarking can extend to hardware and software infrastructure, examining the performance of individual components like GPUs or TPUs.

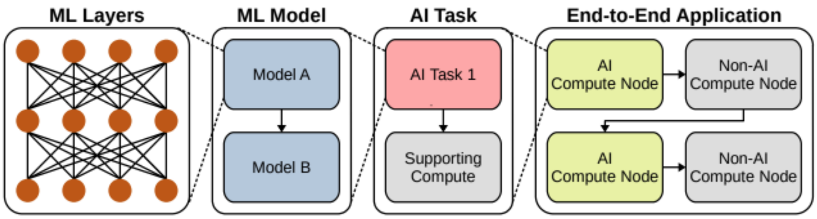


Figure 12.5: ML system granularity.

12.5.1 Micro Benchmarks

Micro-benchmarks are specialized evaluation tools that assess distinct components or specific operations within a broader machine learning process. These benchmarks isolate individual tasks to provide detailed insights into the computational demands of particular system elements, from neural network layers to optimization techniques to activation functions. For example, micro-benchmarks might measure the time required to execute a convolutional layer in a deep learning model or evaluate the speed of data preprocessing operations that prepare training data.

A key area of micro-benchmarking focuses on tensor operations, which are the computational foundation of deep learning. Libraries like cuDNN by NVIDIA provide benchmarks for measuring fundamental computations such as convolutions and matrix multiplications across different hardware configurations. These measurements help developers understand how their hardware handles the core mathematical operations that dominate ML workloads.

Micro-benchmarks also examine activation functions and neural network layers in isolation. This includes measuring the performance of various activation functions like ReLU, Sigmoid, and Tanh under controlled conditions, as well as evaluating the computational efficiency of distinct neural network components such as LSTM cells or Transformer blocks when processing standardized inputs.

DeepBench, developed by Baidu, was one of the first to demonstrate the value of comprehensive micro-benchmarking. It evaluates these fundamental operations across different hardware platforms, providing detailed performance data that helps developers optimize their deep learning implementations. By isolating and measuring individual operations, DeepBench enables precise comparison of hardware platforms and identification of potential performance bottlenecks.

12.5.2 Macro Benchmarks

While micro-benchmarks examine individual operations like tensor computations and layer performance, macro benchmarks evaluate complete machine learning models. This shift from component-level to model-level assessment provides insights into how architectural choices and component interactions affect overall model behavior. For instance, while micro-benchmarks might show optimal performance for individual convolutional layers, macro-benchmarks reveal how these layers work together within a complete convolutional neural network.

Macro-benchmarks measure multiple performance dimensions that emerge only at the model level. These include prediction accuracy, which shows how well the model generalizes to new data; memory consumption patterns across different batch sizes and sequence lengths; throughput under varying computational loads; and latency across different hardware configurations. Understanding these metrics helps developers make informed decisions about model architecture, optimization strategies, and deployment configurations.

The assessment of complete models occurs under standardized conditions using established datasets and tasks. For example, computer vision models might be evaluated on ImageNet, measuring both computational efficiency and prediction accuracy. Natural language processing models might be assessed on translation tasks, examining how they balance quality and speed across different language pairs.

Several industry-standard benchmarks enable consistent model evaluation across platforms. MLPerf Inference provides comprehensive testing suites adapted for different computational environments (Reddi et al. 2019). MLPerf Mobile focuses on mobile device constraints (Janapa Reddi et al. 2022), while MLPerf Tiny addresses microcontroller deployments (C. Banbury et al. 2021). For embedded systems, EEMBC's MLMark[4] emphasizes both performance and power efficiency. The AI-Benchmark suite specializes in mobile platforms, evaluating models across diverse tasks from image recognition to face parsing.

12.5.3 End-to-End Benchmarks

End-to-end benchmarks provide an all-inclusive evaluation that extends beyond the boundaries of the ML model itself. Rather than focusing solely on a machine learning model's computational efficiency or accuracy, these benchmarks encompass the entire pipeline of an AI system. This includes initial ETL (Extract-Transform-Load) or ELT (Extract-Load-Transform) data processing, the core model's performance, post-processing of results, and critical infrastructure components like storage and network systems.

Data processing is the foundation of all AI systems, transforming raw data into a format suitable for model training or inference. In ETL pipelines, data undergoes extraction from source systems, transformation through cleaning and feature engineering, and loading into model-ready formats. These preprocessing steps' efficiency, scalability, and accuracy significantly impact overall system performance. End-to-end benchmarks must assess standardized datasets through these pipelines to ensure data preparation doesn't become a bottleneck.

The post-processing phase plays an equally important role. This involves interpreting the model's raw outputs, converting scores into meaningful categories, filtering results based on predefined tasks, or integrating with other systems. For instance, a computer vision system might need to post-process detection boundaries, apply confidence thresholds, and format results for downstream applications. In real-world deployments, this phase proves crucial for delivering actionable insights.

Beyond core AI operations, infrastructure components heavily influence overall performance and user experience. Storage solutions, whether cloud-based, on-premises, or hybrid, can significantly impact data retrieval and storage times, especially with vast AI datasets. Network interactions, vital for distributed systems, can become performance bottlenecks if not optimized. End-to-end benchmarks must evaluate these components under specified environmental conditions to ensure reproducible measurements of the entire system.

To date, there are no public, end-to-end benchmarks that fully account for data storage, network, and compute performance. While MLPerf Training and Inference approach end-to-end evaluation, they primarily focus on model performance rather than real-world deployment scenarios. Nonetheless, they provide valuable baseline metrics for assessing AI system capabilities.

[4] EEMBC (Embedded Microprocessor Benchmark Consortium): A nonprofit industry group that develops benchmarks for embedded systems, including MLMark for evaluating machine learning workloads.

Given the inherent specificity of end-to-end benchmarking, organizations typically perform these evaluations internally by instrumenting production deployments. This allows engineers to develop result interpretation guidelines based on realistic workloads, but given the sensitivity and specificity of the information, these benchmarks rarely appear in public settings.

12.5.4 Trade-offs

As shown in Table 12.1, different challenges emerge at different stages of an AI system's lifecycle. Each benchmarking approach provides unique insights: micro-benchmarks help engineers optimize specific components like GPU kernel implementations or data loading operations, macro-benchmarks guide model architecture decisions and algorithm selection, while end-to-end benchmarks reveal system-level bottlenecks in production environments.

Table 12.1: Comparison of benchmarking approaches across different dimensions. Each approach offers distinct advantages and focuses on different aspects of ML system evaluation.

Component	Micro Benchmarks	Macro Benchmarks	End-to-End Benchmarks
Focus	Individual operations	Complete models	Full system pipeline
Scope	Tensor ops, layers, activations	Model architecture, training, inference	ETL, model, infrastructure
Example	Conv layer performance on cuDNN	ResNet-50 on ImageNet	Production recommendation system
Advantages	Precise bottleneck identification, Component optimization	Model architecture comparison, Standardized evaluation	Realistic performance assessment, System-wide insights
Challenges	May miss interaction effects	Limited infrastructure insights	Complex to standardize, Often proprietary
Typical Use	Hardware selection, Operation optimization	Model selection, Research comparison	Production system evaluation

Component interaction often produces unexpected behaviors. For example, while micro-benchmarks might show excellent performance for individual convolutional layers, and macro-benchmarks might demonstrate strong accuracy for the complete model, end-to-end evaluation could reveal that data preprocessing creates unexpected bottlenecks during high-traffic periods. These system-level insights often remain hidden when components undergo isolated testing.

Component interaction often produces unexpected behaviors. For example, while micro-benchmarks might show excellent performance for individual convolutional layers, and macro-benchmarks might demonstrate strong accuracy for the complete model, end-to-end evaluation could reveal that data preprocessing creates unexpected bottlenecks during high-traffic periods. These system-level insights often remain hidden when components undergo isolated testing.

12.6 Training Benchmarks

Training benchmarks provide a systematic approach to evaluating the efficiency, scalability, and resource demands of the training phase. They allow practitioners to assess how different design choices, including model architectures, data loading mechanisms, hardware configurations, and distributed training strategies, impact performance. These benchmarks are particularly vital as machine learning systems grow in scale, requiring billions of parameters, terabytes of data, and distributed computing environments.

For instance, large-scale models like OpenAI's GPT-3 (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, and al. 2020), which consists of 175 billion parameters trained on 45 terabytes of data, highlight the immense computational demands of training. Benchmarks enable systematic evaluation of the underlying systems to ensure that hardware and software configurations can meet these demands efficiently.

ℹ Definition of ML Training Benchmarks

ML Training Benchmarks are standardized tools used to evaluate the *performance*, *efficiency*, and *scalability* of machine learning systems during the *training phase*. These benchmarks measure key *system-level metrics*, such as *time-to-accuracy*, *throughput*, *resource utilization*, and *energy consumption*. By providing a structured evaluation framework, training benchmarks enable *fair comparisons* across *hardware platforms*, *software frameworks*, and *distributed computing setups*. They help identify *bottlenecks* and optimize *training processes* for *large-scale machine learning models*, ensuring that computational resources are used effectively.

Efficient data storage and delivery during training also play a major role in the training process. For instance, in a machine learning model that predicts bounding boxes around objects in an image, thousands of images may be required. However, loading an entire image dataset into memory is typically infeasible, so practitioners rely on data loaders from ML frameworks. Successful model training depends on timely and efficient data delivery, making it essential to benchmark tools like data pipelines, preprocessing speed, and storage retrieval times to understand their impact on training performance.

Hardware selection is another key factor in training machine learning systems, as it can significantly impact training time. Training benchmarks evaluate CPU, GPU, memory, and network utilization during the training phase to guide system optimizations. Understanding how resources are used is essential: Are GPUs being fully leveraged? Is there unnecessary memory overhead? Benchmarks can uncover bottlenecks or inefficiencies in resource utilization, leading to cost savings and performance improvements.

In many cases, using a single hardware accelerator, such as a single GPU, is insufficient to meet the computational demands of large-scale model training. Machine learning models are often trained in data centers with multiple GPUs or TPUs, where distributed computing enables parallel processing across nodes. Training benchmarks assess how efficiently the system scales across multiple nodes, manages data sharding, and handles challenges like node failures or drop-offs during training.

To illustrate these benchmarking principles, we will reference MLPerf Training throughout this section. Briefly, MLPerf is an industry-standard benchmark suite designed to evaluate machine learning system performance. It provides standardized tests for training and inference across a range of deep learning workloads, including image classification, language modeling, object detection, and recommendation systems.

12.6.1 Motivation

From a systems perspective, training machine learning models is a computationally intensive process that requires careful optimization of resources. Training benchmarks serve as essential tools for evaluating system efficiency, identifying bottlenecks, and ensuring that machine learning systems can scale effectively. They provide a standardized approach to measuring how various system components, including hardware accelerators, memory, storage, and network infrastructure, affect training performance.

Training benchmarks enable researchers and engineers to push the state-of-the-art, optimize configurations, improve scalability, and reduce overall resource consumption by systematically evaluating these factors. As shown in Figure 12.6, the performance improvements in progressive versions of MLPerf Training benchmarks have consistently outpaced Moore's Law, which demonstrates that what gets measured gets improved. Using standardized benchmarking trends allows us to rigorously showcase the rapid evolution of ML computing.

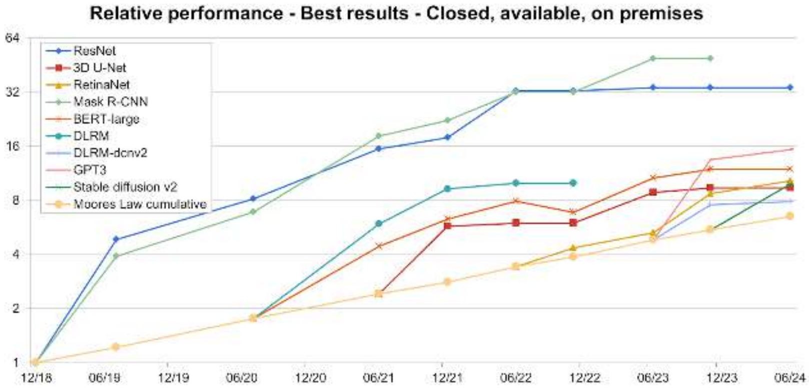


Figure 12.6: MLPerf Training performance trends. Source: Tschand et al. (2024).

12.6.1.1 Importance of Training Benchmarks

As machine learning models grow in complexity, training becomes increasingly demanding in terms of compute power, memory, and data storage. The ability to measure and compare training efficiency is critical to ensuring that systems can effectively handle large-scale workloads. Training benchmarks provide a structured methodology for assessing performance across different hardware platforms, software frameworks, and optimization techniques.

One of the fundamental challenges in training machine learning models is the efficient allocation of computational resources. Training a transformer-based model such as GPT-3, which consists of 175 billion parameters and requires processing terabytes of data, places an enormous burden on modern computing infrastructure. Without standardized benchmarks, it becomes difficult to determine whether a system is fully utilizing its resources or whether inefficiencies, including slow data loading, underutilized accelerators, and excessive memory overhead, are limiting performance.

Training benchmarks help uncover such inefficiencies by measuring key performance indicators, including system throughput, time-to-accuracy, and hardware utilization. These benchmarks allow practitioners to analyze whether GPUs, TPUs, and CPUs are being leveraged effectively or whether specific bottlenecks, such as memory bandwidth constraints or inefficient data pipelines, are reducing overall system performance. For example, a system using TF32[5] precision1 may achieve higher throughput than one using FP32, but if TF32 introduces numerical instability that increases the number of iterations required to reach the target accuracy, the overall training time may be longer. By providing insights into these factors, benchmarks support the design of more efficient training workflows that maximize hardware potential while minimizing unnecessary computation.

12.6.1.2 Hardware & Software Optimization

The performance of machine learning training is heavily influenced by the choice of hardware and software. Training benchmarks guide system designers in selecting optimal configurations by measuring how different architectures, including GPUs, TPUs, and emerging AI accelerators, handle computational workloads. These benchmarks also evaluate how well deep learning frameworks, such as TensorFlow and PyTorch, optimize performance across different hardware setups.

For example, the MLPerf Training benchmark suite is widely used to compare the performance of different accelerator architectures on tasks such as image classification, natural language processing, and recommendation systems. By running standardized benchmarks across multiple hardware configurations, engineers can determine whether certain accelerators are better suited for specific training workloads. This information is particularly valuable in large-scale data centers and cloud computing environments, where selecting the right combination of hardware and software can lead to significant performance gains and cost savings.

Beyond hardware selection, training benchmarks also inform software optimizations. Machine learning frameworks implement various low-level optimizations, including mixed-precision training, memory-efficient data loading, and distributed training strategies, that can significantly impact system performance. Benchmarks help quantify the impact of these optimizations, ensuring that training systems are configured for maximum efficiency.

12.6.1.3 Scalability & Efficiency

As machine learning workloads continue to grow, efficient scaling across distributed computing environments has become a key concern. Many modern

[5] TensorFloat-32 (TF32): Introduced in NVIDIA Ampere GPUs, provides higher throughput than FP32 but may introduce numerical stability issues affecting model convergence.

deep learning models are trained across multiple GPUs or TPUs, requiring efficient parallelization strategies to ensure that additional computing resources lead to meaningful performance improvements. Training benchmarks measure how well a system scales by evaluating system throughput, memory efficiency, and overall training time as additional computational resources are introduced.

Effective scaling is not always guaranteed. While adding more GPUs or TPUs should, in theory, reduce training time, issues such as communication overhead, data synchronization latency, and memory bottlenecks can limit scaling efficiency. Training benchmarks help identify these challenges by quantifying how performance scales with increasing hardware resources. A well-designed system should exhibit near-linear scaling, where doubling the number of GPUs results in a near-halving of training time. However, real-world inefficiencies often prevent perfect scaling, and benchmarks provide the necessary insights to optimize system design accordingly.

Another crucial factor in training efficiency is time-to-accuracy, which measures how quickly a model reaches a target accuracy level. Achieving faster convergence with fewer computational resources is a key goal in training optimization, and benchmarks help compare different training methodologies to determine which approaches strike the best balance between speed and accuracy. By leveraging training benchmarks, system designers can assess whether their infrastructure is capable of handling large-scale workloads efficiently while maintaining training stability and accuracy.

12.6.1.4 Cost & Energy Factors

The computational cost of training large-scale models has risen sharply in recent years, making cost-efficiency a critical consideration. Training a model such as GPT-3 can require millions of dollars in cloud computing resources, making it imperative to evaluate cost-effectiveness across different hardware and software configurations. Training benchmarks provide a means to quantify the cost per training run by analyzing computational expenses, cloud pricing models, and energy consumption.

Beyond financial cost, energy efficiency has become an increasingly important metric. Large-scale training runs consume vast amounts of electricity, contributing to significant carbon emissions. Benchmarks help evaluate energy efficiency by measuring power consumption per unit of training progress, allowing organizations to identify sustainable approaches to AI development.

For example, MLPerf includes an energy benchmarking component that tracks the power consumption of various hardware accelerators during training. This allows researchers to compare different computing platforms not only in terms of raw performance but also in terms of their environmental impact. By integrating energy efficiency metrics into benchmarking studies, organizations can design AI systems that balance computational power with sustainability goals.

12.6.1.5 Fair ML Systems Comparison

One of the primary functions of training benchmarks is to establish a standardized framework for comparing ML systems. Given the wide variety of hardware architectures, deep learning frameworks, and optimization techniques available today, ensuring fair and reproducible comparisons is essential.

Standardized benchmarks provide a common evaluation methodology, allowing researchers and practitioners to assess how different training systems perform under identical conditions. For example, MLPerf Training benchmarks enable vendor-neutral comparisons by defining strict evaluation criteria for deep learning tasks such as image classification, language modeling, and recommendation systems. This ensures that performance results are meaningful and not skewed by differences in dataset preprocessing, hyperparameter tuning, or implementation details.

Furthermore, reproducibility is a major concern in machine learning research. Training benchmarks help address this challenge by providing clearly defined methodologies for performance evaluation, ensuring that results can be consistently reproduced across different computing environments. By adhering to standardized benchmarks, researchers can make informed decisions when selecting hardware, software, and training methodologies, ultimately driving progress in AI systems development.

12.6.2 Metrics

Evaluating the performance of machine learning training requires a set of well-defined metrics that go beyond conventional algorithmic measures. From a systems perspective, training benchmarks assess how efficiently and effectively a machine learning model can be trained to a predefined accuracy threshold. Metrics such as throughput, scalability, and energy efficiency are only meaningful in relation to whether the model successfully reaches its target accuracy. Without this constraint, optimizing for raw speed or resource utilization may lead to misleading conclusions.

Training benchmarks, such as MLPerf Training, define specific accuracy targets for different machine learning tasks, ensuring that performance measurements are made in a fair and reproducible manner. A system that trains a model quickly but fails to reach the required accuracy is not considered a valid benchmark result. Conversely, a system that achieves the best possible accuracy but takes an excessive amount of time or resources may not be practically useful. Effective benchmarking requires balancing speed, efficiency, and accuracy convergence.

12.6.2.1 Time and Throughput

One of the fundamental metrics for evaluating training efficiency is the time required to reach a predefined accuracy threshold. Training time (T_{train}) measures how long a model takes to converge to an acceptable performance level, reflecting the overall computational efficiency of the system. It is formally defined as:

$$T_{\text{train}} = \arg\min_{t} \{\text{accuracy}(t) \geq \text{target accuracy}\}$$

This metric ensures that benchmarking focuses on how quickly and effectively a system can achieve meaningful results.

Throughput, often expressed as the number of training samples processed per second, provides an additional measure of system performance:

$$T = \frac{N_{\text{samples}}}{T_{\text{train}}}$$

where N_{samples} is the total number of training samples processed. However, throughput alone does not guarantee meaningful results, as a model may process a large number of samples quickly without necessarily reaching the desired accuracy.

For example, in MLPerf Training, the benchmark for ResNet-50 may require reaching an accuracy target like 75.9% top-1 on the ImageNet dataset. A system that processes 10,000 images per second but fails to achieve this accuracy is not considered a valid benchmark result, while a system that processes fewer images per second but converges efficiently is preferable. This highlights why throughput must always be evaluated in relation to time-to-accuracy rather than as an independent performance measure.

12.6.2.2 Scalability & Parallelism

As machine learning models increase in size, training workloads often require distributed computing across multiple processors or accelerators. Scalability measures how effectively training performance improves as more computational resources are added. An ideal system should exhibit near-linear scaling, where doubling the number of GPUs or TPUs leads to a proportional reduction in training time. However, real-world performance is often constrained by factors such as communication overhead, memory bandwidth limitations, and inefficiencies in parallelization strategies.

When training large-scale models such as GPT-3, OpenAI employed thousands of GPUs in a distributed training setup. While increasing the number of GPUs provided more raw computational power, the performance improvements were not perfectly linear due to network communication overhead between nodes. Benchmarks such as MLPerf quantify how well a system scales across multiple GPUs, providing insights into where inefficiencies arise in distributed training.

Parallelism in training is categorized into data parallelism, model parallelism, and pipeline parallelism, each presenting distinct challenges. Data parallelism, the most commonly used strategy, involves splitting the training dataset across multiple compute nodes. The efficiency of this approach depends on synchronization mechanisms and gradient communication overhead. In contrast, model parallelism partitions the neural network itself, requiring efficient coordination between processors. Benchmarks evaluate how well a system manages these parallelism strategies without degrading accuracy convergence.

12.6.2.3 Resource Utilization

The efficiency of machine learning training depends not only on speed and scalability but also on how well available hardware resources are utilized. Compute utilization measures the extent to which processing units, such as

GPUs or TPUs, are actively engaged during training. Low utilization may indicate bottlenecks in data movement, memory access, or inefficient workload scheduling.

For instance, when training BERT on a TPU cluster, researchers observed that input pipeline inefficiencies were limiting overall throughput. Although the TPUs had high raw compute power, the system was not keeping them fully utilized due to slow data retrieval from storage. By profiling the resource utilization, engineers identified the bottleneck and optimized the input pipeline using TFRecord and data prefetching, leading to improved performance.

Memory bandwidth is another critical factor, as deep learning models require frequent access to large volumes of data during training. If memory bandwidth becomes a limiting factor, increasing compute power alone will not improve training speed. Benchmarks assess how well models leverage available memory, ensuring that data transfer rates between storage, main memory, and processing units do not become performance bottlenecks.

I/O performance also plays a significant role in training efficiency, particularly when working with large datasets that cannot fit entirely in memory. Benchmarks evaluate the efficiency of data loading pipelines, including preprocessing operations, caching mechanisms, and storage retrieval speeds. Systems that fail to optimize data loading can experience significant slowdowns, regardless of computational power.

12.6.2.4 Energy Efficiency & Cost

Training large-scale machine learning models requires substantial computational resources, leading to significant energy consumption and financial costs. Energy efficiency metrics quantify the power usage of training workloads, helping identify systems that optimize computational efficiency while minimizing energy waste. The increasing focus on sustainability has led to the inclusion of energy-based benchmarks, such as those in MLPerf Training, which measure power consumption per training run.

Training GPT-3 was estimated to consume 1,287 MWh of electricity, which is comparable to the yearly energy usage of 100 US households. If a system can achieve the same accuracy with fewer training iterations, it directly reduces energy consumption. Energy-aware benchmarks help guide the development of hardware and training strategies that optimize power efficiency while maintaining accuracy targets.

Cost considerations extend beyond electricity usage to include hardware expenses, cloud computing costs, and infrastructure maintenance. Training benchmarks provide insights into the cost-effectiveness of different hardware and software configurations by measuring training time in relation to resource expenditure. Organizations can use these benchmarks to balance performance and budget constraints when selecting training infrastructure.

12.6.2.5 Fault Tolerance & Robustness

Training workloads often run for extended periods, sometimes spanning days or weeks, making fault tolerance an essential consideration. A robust system must be capable of handling unexpected failures, including hardware malfunctions, network disruptions, and memory errors, without compromising accuracy convergence.

In large-scale cloud-based training, node failures are common due to hardware instability. If a GPU node in a distributed cluster fails, training must continue without corrupting the model. MLPerf Training includes evaluations of fault-tolerant training strategies, such as checkpointing, where models periodically save their progress. This ensures that failures do not require restarting the entire training process.

12.6.2.6 Reproducibility & Standardization

For benchmarks to be meaningful, results must be reproducible across different runs, hardware platforms, and software frameworks. Variability in training results can arise due to stochastic processes, hardware differences, and software optimizations. Ensuring reproducibility requires standardizing evaluation protocols, controlling for randomness in model initialization, and enforcing consistency in dataset processing.

MLPerf Training enforces strict reproducibility requirements, ensuring that accuracy results remain stable across multiple training runs. When NVIDIA submitted benchmark results for MLPerf, they had to demonstrate that their ResNet-50 ImageNet training time remained consistent across different GPUs. This ensures that benchmarks measure true system performance rather than noise from randomness.

12.6.3 Training Performance Evaluation

Evaluating the performance of machine learning training systems involves more than just measuring how fast a model can be trained. A comprehensive benchmarking approach considers multiple dimensions—each capturing a different aspect of system behavior. The specific metrics used depend on the goals of the evaluation, whether those are optimizing speed, improving resource efficiency, reducing energy consumption, or ensuring robustness and reproducibility.

Table 12.2 provides an overview of the core categories and associated metrics commonly used to benchmark system-level training performance. These categories serve as a framework for understanding how training systems behave under different workloads and configurations.

Table 12.2: Training benchmark metrics and evaluation dimensions.

Category	Key Metrics	Example Benchmark Use
Training Time and Throughput	Time-to-accuracy (seconds, minutes, hours); Throughput (samples/sec)	Comparing training speed across different GPU architectures
Scalability and Parallelism	Scaling efficiency (% of ideal speedup); Communication overhead (latency, bandwidth)	Analyzing distributed training performance for large models
Resource Utilization	Compute utilization (% GPU/TPU usage); Memory bandwidth (GB/s); I/O efficiency (data loading speed)	Optimizing data pipelines to improve GPU utilization
Energy Efficiency and Cost	Energy consumption per run (MWh, kWh); Performance per watt (TOPS/W)	Evaluating energy-efficient training strategies
Fault Tolerance and Robustness	Checkpoint overhead (time per save); Recovery success rate (%)	Assessing failure recovery in cloud-based training systems

Category	Key Metrics	Example Benchmark Use
Reproducibility and Standardization	Variance across runs (% difference in accuracy, training time); Framework consistency (TensorFlow vs. PyTorch vs. JAX)	Ensuring consistency in benchmark results across hardware

Training time and throughput are often the first metrics considered when evaluating system performance. Time-to-accuracy, the duration required for a model to achieve a specified accuracy level, is a practical and widely used benchmark. Throughput, typically measured in samples per second, provides insight into how efficiently data is processed during training. For example, when comparing a ResNet-50 model trained on NVIDIA A100 versus V100 GPUs, the A100 generally offers higher throughput and faster convergence. However, it is important to ensure that increased throughput does not come at the expense of convergence quality, especially when reduced numerical precision (e.g., TF32) is used to speed up computation.

As model sizes continue to grow, scalability becomes a critical performance dimension. Efficient use of multiple GPUs or TPUs is essential for training large models such as GPT-3 or T5. In this context, scaling efficiency and communication overhead are key metrics. A system might scale linearly up to 64 GPUs, but beyond that, performance gains may taper off due to increased synchronization and communication costs. Benchmarking tools that monitor interconnect bandwidth and gradient aggregation latency can reveal how well a system handles distributed training.

Resource utilization complements these measures by examining how effectively a system leverages its compute and memory resources. Metrics such as GPU utilization, memory bandwidth, and data loading efficiency help identify performance bottlenecks. For instance, a BERT pretraining task that exhibits only moderate GPU utilization may be constrained by an underperforming data pipeline. Optimizations like sharding input files or prefetching data into device memory can often resolve these inefficiencies.

In addition to raw performance, energy efficiency and cost have become increasingly important considerations. Training large models at scale can consume significant power, raising environmental and financial concerns. Metrics such as energy consumed per training run and performance per watt (e.g., TOPS/W) help evaluate the sustainability of different hardware and system configurations. For example, while two systems may reach the same accuracy in the same amount of time, the one that uses significantly less energy may be preferred for long-term deployment.

Fault tolerance and robustness address how well a system performs under non-ideal conditions, which are common in real-world deployments. Training jobs frequently encounter hardware failures, preemptions, or network instability. Metrics like checkpoint overhead and recovery success rate provide insight into the resilience of a training system. In practice, checkpointing can introduce non-trivial overhead—for example, pausing training every 30 minutes to write a full checkpoint may reduce overall throughput by 5-10%. Systems must strike a balance between failure recovery and performance impact.

Finally, reproducibility and standardization ensure that benchmark results are consistent, interpretable, and transferable. Even minor differences in software libraries, initialization seeds, or floating-point behavior can affect training outcomes. Comparing the same model across frameworks, such as comparing PyTorch with Automatic Mixed Precision to TensorFlow with XLA, can reveal variation in convergence rates or final accuracy. Reliable benchmarking requires careful control of these variables, along with repeated runs to assess statistical variance.

Together, these dimensions provide a holistic view of training performance. They help researchers, engineers, and system designers move beyond simplistic comparisons and toward a more nuanced understanding of how machine learning systems behave under realistic conditions. As training workloads continue to scale, such multidimensional evaluation will be essential for guiding hardware choices, software optimizations, and infrastructure design.

12.6.3.1 Training Benchmark Pitfalls

Despite the availability of well-defined benchmarking methodologies, certain misconceptions and flawed evaluation practices often lead to misleading conclusions. Understanding these pitfalls is important for interpreting benchmark results correctly.

Overemphasis on Raw Throughput. A common mistake in training benchmarks is assuming that higher throughput always translates to better training performance. It is possible to artificially increase throughput by using lower numerical precision, reducing synchronization, or even bypassing certain computations. However, these optimizations do not necessarily lead to faster convergence.

For example, a system using TF32 precision may achieve higher throughput than one using FP32, but if TF32 introduces numerical instability that increases the number of iterations required to reach the target accuracy, the overall training time may be longer. The correct way to evaluate throughput is in relation to time-to-accuracy, ensuring that speed optimizations do not come at the expense of convergence efficiency.

Isolated Single-Node Performance. Benchmarking training performance on a single node without considering how well it scales in a distributed setting can lead to misleading conclusions. A GPU may demonstrate excellent throughput when used independently, but when deployed across hundreds of nodes, communication overhead and synchronization constraints may diminish these efficiency gains.

For instance, a system optimized for single-node performance may employ memory optimizations that do not generalize well to multi-node environments. Large-scale models such as GPT-3 require efficient gradient synchronization across multiple nodes, making it essential to assess scalability rather than relying solely on single-node performance metrics.

Ignoring Failures & Interference. Many benchmarks assume an idealized training environment where hardware failures, memory corruption, network instability, or interference from other processes do not occur. However, real-world training jobs often experience unexpected failures and workload interference that require checkpointing, recovery mechanisms, and resource management.

A system optimized for ideal-case performance but lacking fault tolerance and interference handling may achieve impressive benchmark results under controlled conditions, but frequent failures, inefficient recovery, and resource contention could make it impractical for large-scale deployment. Effective benchmarking should consider checkpointing overhead, failure recovery efficiency, and the impact of interference from other processes rather than assuming perfect execution conditions.

Linear Scaling Assumption. When evaluating distributed training, it is often assumed that increasing the number of GPUs or TPUs will result in proportional speedups. In practice, communication bottlenecks, memory contention, and synchronization overheads lead to diminishing returns as more compute nodes are added.

For example, training a model across 1,000 GPUs does not necessarily provide 100 times the speed of training on 10 GPUs. At a certain scale, gradient communication costs become a limiting factor, offsetting the benefits of additional parallelism. Proper benchmarking should assess scalability efficiency rather than assuming idealized linear improvements.

Ignoring Reproducibility. Benchmark results are often reported without verifying their reproducibility across different hardware and software frameworks. Even minor variations in floating-point arithmetic, memory layouts, or optimization strategies can introduce statistical differences in training time and accuracy.

For example, a benchmark run on TensorFlow with XLA optimizations may exhibit different convergence characteristics compared to the same model trained using PyTorch with Automatic Mixed Precision (AMP). Proper benchmarking requires evaluating results across multiple frameworks to ensure that software-specific optimizations do not distort performance comparisons.

12.6.3.2 Final Thoughts

Training benchmarks provide valuable insights into machine learning system performance, but their interpretation requires careful consideration of real-world constraints. High throughput does not necessarily mean faster training if it compromises accuracy convergence. Similarly, scaling efficiency must be evaluated holistically, taking into account both computational efficiency and communication overhead.

Avoiding common benchmarking pitfalls and employing structured evaluation methodologies allows machine learning practitioners to gain a deeper understanding of how to optimize training workflows, design efficient AI systems, and develop scalable machine learning infrastructure. As models continue to increase in complexity, benchmarking methodologies must evolve to reflect real-world challenges, ensuring that benchmarks remain meaningful and actionable in guiding AI system development.

12.7 Inference Benchmarks

Inference benchmarks provide a systematic approach to evaluating the efficiency, latency, and resource demands of the inference phase in machine learning systems. Unlike training, where the focus is on optimizing large-scale computations over extensive datasets, inference involves deploying trained models to make real-time or batch predictions efficiently. These benchmarks help assess how various factors, including model architectures, hardware configurations, quantization techniques, and runtime optimizations, impact inference performance.

As deep learning models grow in complexity and size, efficient inference becomes a key challenge, particularly for applications requiring real-time decision-making, such as autonomous driving, healthcare diagnostics, and conversational AI. For example, serving large-scale models like OpenAI's GPT-4 involves handling billions of parameters while maintaining low latency. Inference benchmarks enable systematic evaluation of the underlying hardware and software stacks to ensure that models can be deployed efficiently across different environments, from cloud data centers to edge devices.

ℹ Definition of ML Inference Benchmarks

ML Inference Benchmarks are standardized tools used to evaluate the *performance*, *efficiency*, and *scalability* of machine learning systems during the *inference phase*. These benchmarks measure key *system-level metrics*, such as *latency*, *throughput*, *energy consumption*, and *memory footprint*. By providing a structured evaluation framework, inference benchmarks enable *fair comparisons* across *hardware platforms*, *software runtimes*, and *deployment configurations*. They help identify *bottlenecks* and optimize *inference pipelines* for *real-time and large-scale machine learning applications*, ensuring that computational resources are utilized effectively.

Unlike training, which is often conducted in large-scale data centers with ample computational resources, inference must be optimized for diverse deployment scenarios, including mobile devices, IoT systems, and embedded processors. Efficient inference depends on multiple factors, such as optimized data pipelines, quantization, pruning, and hardware acceleration. Benchmarks help evaluate how well these optimizations improve real-world deployment performance.

Hardware selection plays an important role in inference efficiency. While GPUs and TPUs are widely used for training, inference workloads often require specialized accelerators like NPUs (Neural Processing Units), FPGAs, and dedicated inference chips such as Google's Edge TPU. Inference benchmarks evaluate the utilization and performance of these hardware components, helping practitioners choose the right configurations for their deployment needs.

Scaling inference workloads across cloud servers, edge platforms, mobile devices, and tinyML systems introduces additional challenges. As illustrated in Figure 12.7, there is a significant differential in power consumption among these systems, ranging from microwatts to megawatts. Inference benchmarks evaluate the trade-offs between latency, cost, and energy efficiency, thereby assisting organizations in making informed deployment decisions.

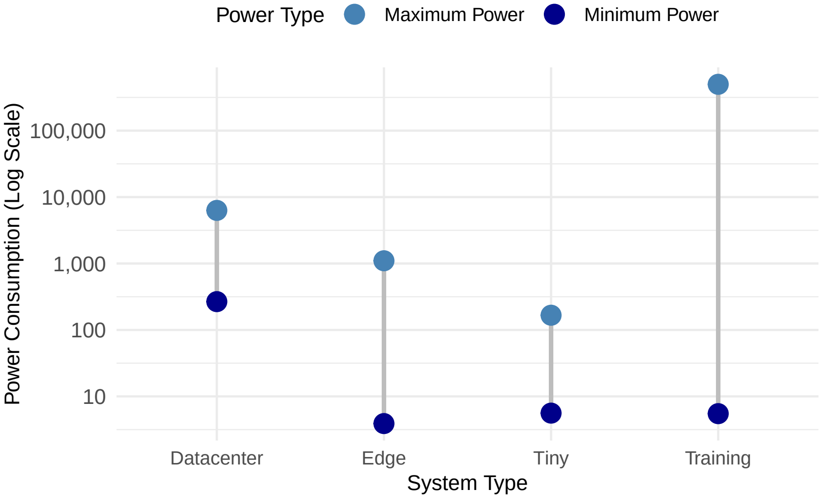


Figure 12.7: Energy consumption by system type.

As with training, we will reference MLPerf Inference throughout this section to illustrate benchmarking principles. MLPerf provides standardized inference tests across different workloads, including image classification, object detection, speech recognition, and language processing. A full discussion of MLPerf's methodology and structure is presented later in this chapter.

12.7.1 Motivation

Deploying machine learning models for inference introduces a unique set of challenges distinct from training. While training optimizes large-scale computation over extensive datasets, inference must deliver predictions efficiently and at scale in real-world environments. Inference benchmarks provide a systematic approach to evaluating system performance, identifying bottlenecks, and ensuring that models can operate effectively across diverse deployment scenarios.

Unlike training, which typically runs on dedicated high-performance hardware, inference must adapt to varying constraints. A model deployed in a cloud server might prioritize high-throughput batch processing, while the same model running on a mobile device must operate under strict latency and power constraints. On edge devices with limited compute and memory, optimizations such as quantization and pruning become critical. Benchmarks help assess these trade-offs, ensuring that inference systems maintain the right balance between accuracy, speed, and efficiency across different platforms.

Inference benchmarks help answer fundamental questions about model deployment. How quickly can a model generate predictions in real-world conditions? What are the trade-offs between inference speed and accuracy? Can an inference system handle increasing demand while maintaining low latency? By evaluating these factors, benchmarks guide optimizations in both hardware and software to improve overall efficiency (Reddi et al. 2019).

12.7.1.1 Importance of Inference Benchmarks

Inference plays a critical role in AI applications, where performance directly affects usability and cost. Unlike training, which is often performed offline, inference typically operates in real-time or near real-time, making latency a primary concern. A self-driving car processing camera feeds must react within milliseconds, while a voice assistant generating responses should feel instantaneous to users.

Different applications impose varying constraints on inference. Some workloads require single-instance inference, where predictions must be made as quickly as possible for each individual input. This is crucial in real-time systems such as robotics, augmented reality, and conversational AI, where even small delays can impact responsiveness. Other workloads, such as large-scale recommendation systems or search engines, process massive batches of queries simultaneously, prioritizing throughput over per-query latency. Benchmarks allow engineers to evaluate both scenarios and ensure models are optimized for their intended use case.

A key difference between training and inference is that inference workloads often run continuously in production, meaning that small inefficiencies can compound over time. Unlike a training job that runs once and completes, an inference system deployed in the cloud may serve millions of queries daily, and a model running on a smartphone must manage battery consumption over extended use. Benchmarks provide a structured way to measure inference efficiency under these real-world constraints, helping developers make informed choices about model optimization, hardware selection, and deployment strategies.

12.7.1.2 Hardware & Software Optimization

Efficient inference depends on both hardware acceleration and software optimizations. While GPUs and TPUs dominate training, inference is more diverse in its hardware needs. A cloud-based AI service might leverage powerful accelerators for large-scale workloads, whereas mobile devices rely on specialized inference chips like NPUs or optimized CPU execution. On embedded systems, where resources are constrained, achieving high performance requires careful memory and compute efficiency. Benchmarks help evaluate how well different hardware platforms handle inference workloads, guiding deployment decisions.

Software optimizations are just as important. Frameworks like TensorRT, ONNX Runtime, and TVM apply optimizations such as operator fusion, quantization, and kernel tuning to improve inference speed and reduce computational overhead. These optimizations can make a significant difference, especially in environments with limited resources. Benchmarks allow developers to measure the impact of such techniques on latency, throughput, and power efficiency, ensuring that optimizations translate into real-world improvements without degrading model accuracy.

12.7.1.3 Scalability & Efficiency

Inference workloads vary significantly in their scaling requirements. A cloud-based AI system handling millions of queries per second must ensure that increasing demand does not cause delays, while a mobile application running a model locally must execute quickly even under power constraints. Unlike training, which is typically performed on a fixed set of high-performance machines, inference must scale dynamically based on usage patterns and available computational resources.

Benchmarks evaluate how inference systems scale under different conditions. They measure how well performance holds up under increasing query loads, whether additional compute resources improve inference speed, and how efficiently models run across different deployment environments. Large-scale inference deployments often involve distributed inference servers, where multiple copies of a model process incoming requests in parallel. Benchmarks assess how efficiently this scaling occurs and whether additional resources lead to meaningful improvements in latency and throughput.

Another key factor in inference efficiency is cold-start performance—the time it takes for a model to load and begin processing queries. This is especially relevant for applications that do not run inference continuously but instead load models on demand. Benchmarks help determine whether a system can quickly transition from idle to active execution without significant overhead.

12.7.1.4 Cost & Energy Factors

Because inference workloads run continuously, operational cost and energy efficiency are critical factors. Unlike training, where compute costs are incurred once, inference costs accumulate over time as models are deployed in production. Running an inefficient model at scale can significantly increase cloud compute expenses, while an inefficient mobile inference system can drain battery life quickly. Benchmarks provide insights into cost per inference request, helping organizations optimize for both performance and affordability.

Energy efficiency is also a growing concern, particularly for mobile and edge AI applications. Many inference workloads run on battery-powered devices, where excessive computation can impact usability. A model running on a smartphone, for example, must be optimized to minimize power consumption while maintaining responsiveness. Benchmarks help evaluate inference efficiency per watt, ensuring that models can operate sustainably across different platforms.

12.7.1.5 Fair ML Systems Comparison

With many different hardware platforms and optimization techniques available, standardized benchmarking is essential for fair comparisons. Without well-defined benchmarks, it becomes difficult to determine whether performance gains come from genuine improvements or from optimizations that exploit specific hardware features. Inference benchmarks provide a consistent evaluation methodology, ensuring that comparisons are meaningful and reproducible.

For example, MLPerf Inference defines rigorous evaluation criteria for tasks such as image classification, object detection, and speech recognition, making it possible to compare different systems under controlled conditions. These standardized tests prevent misleading results caused by differences in dataset preprocessing, proprietary optimizations, or vendor-specific tuning. By enforcing reproducibility, benchmarks allow researchers and engineers to make informed decisions when selecting inference frameworks, hardware accelerators, and optimization techniques.

12.7.2 Metrics

Evaluating the performance of inference systems requires a distinct set of metrics from those used for training. While training benchmarks emphasize throughput, scalability, and time-to-accuracy, inference benchmarks must focus on latency, efficiency, and resource utilization in practical deployment settings. These metrics ensure that machine learning models perform well across different environments, from cloud data centers handling millions of requests to mobile and edge devices operating under strict power and memory constraints.

Unlike training, where the primary goal is to optimize learning speed, inference benchmarks evaluate how efficiently a trained model can process inputs and generate predictions at scale. The following sections describe the most important inference benchmarking metrics, explaining their relevance and how they are used to compare different systems.

12.7.2.1 Latency & Tail Latency

Latency is one of the most critical performance metrics for inference, particularly in real-time applications where delays can negatively impact user experience or system safety. Latency refers to the time taken for an inference system to process an input and produce a prediction. While the average latency of a system is useful, it does not capture performance in high-demand scenarios where occasional delays can degrade reliability.

To account for this, benchmarks often measure tail latency, which reflects the worst-case delays in a system. These are typically reported as the 95th percentile (p95) or 99th percentile (p99) latency, meaning that 95% or 99% of inferences are completed within a given time. For applications such as autonomous driving or real-time trading, maintaining low tail latency is essential to avoid unpredictable delays that could lead to catastrophic outcomes.

12.7.2.2 Throughput & Batch Processing Efficiency

While latency measures the speed of individual inference requests, throughput measures how many inference requests a system can process per second. It is typically expressed in queries per second (QPS) or frames per second (FPS) for vision tasks. Some inference systems operate on a single-instance basis, where each input is processed independently as soon as it arrives. Other systems process multiple inputs in parallel using batch inference, which can significantly improve efficiency by leveraging hardware optimizations.

For example, cloud-based services handling millions of queries per second benefit from batch inference, where large groups of inputs are processed together to maximize computational efficiency. In contrast, applications like robotics, interactive AI, and augmented reality require low-latency single-instance inference, where the system must respond immediately to each new input.

Benchmarks must consider both single-instance and batch throughput to provide a comprehensive understanding of inference performance across different deployment scenarios.

12.7.2.3 Precision & Accuracy Trade-offs

Optimizing inference performance often involves reducing numerical precision, which can significantly accelerate computation while reducing memory and energy consumption. However, lower-precision calculations can introduce accuracy degradation, making it essential to benchmark the trade-offs between speed and predictive quality.

Inference benchmarks evaluate how well models perform under different numerical settings, such as FP32, FP16, and INT8. Many modern AI accelerators support mixed-precision inference, allowing systems to dynamically adjust numerical representation based on workload requirements. Quantization and pruning techniques further improve efficiency, but their impact on model accuracy varies depending on the task and dataset. Benchmarks help determine whether these optimizations are viable for deployment, ensuring that improvements in efficiency do not come at the cost of unacceptable accuracy loss.

12.7.2.4 Memory Footprint & Model Size

Beyond computational optimizations, memory footprint is another critical consideration for inference systems, particularly for devices with limited resources. Efficient inference depends not only on speed but also on memory usage. Unlike training, where large models can be distributed across powerful GPUs or TPUs, inference often requires models to run within strict memory budgets. The total model size determines how much storage is required for deployment, while RAM usage reflects the working memory needed during execution. Some models require large memory bandwidth to efficiently transfer data between processing units, which can become a bottleneck if the hardware lacks sufficient capacity.

Inference benchmarks evaluate these factors to ensure that models can be deployed effectively across a range of devices. A model that achieves high accuracy but exceeds memory constraints may be impractical for real-world use. To address this, compression techniques such as quantization, pruning, and knowledge distillation are often applied to reduce model size while maintaining accuracy. Benchmarks help assess whether these optimizations strike the right balance between memory efficiency and predictive performance.

12.7.2.5 Cold-Start & Model Load Time

Once memory requirements are optimized, cold-start performance[6] becomes critical for ensuring inference systems are ready to respond quickly upon deployment. In many deployment scenarios, models are not always kept in memory

[6] Cold-Start Time: The time required for a model to initialize and become ready to process the first inference request after being loaded from disk or a low-power state.

but instead loaded on demand when needed. This can introduce significant delays, particularly in serverless AI[7] environments, where resources are allocated dynamically based on incoming requests. Cold-start performance measures how quickly a system can transition from idle to active execution, ensuring that inference is available without excessive wait times.

Model load time refers to the duration required to load a trained model into memory before it can process inputs. In some cases, particularly on resource-limited devices, models must be reloaded frequently to free up memory for other applications. The time taken for the first inference request is also an important consideration, as it reflects the total delay users experience when interacting with an AI-powered service. Benchmarks help quantify these delays, ensuring that inference systems can meet real-world responsiveness requirements.

12.7.2.6 Scalability & Dynamic Workload Handling

While cold-start latency addresses initial responsiveness, scalability ensures that inference systems can handle fluctuating workloads and concurrent demands over time Inference workloads must scale effectively across different usage patterns. In cloud-based AI services, this means efficiently handling millions of concurrent users, while on mobile or embedded devices, it involves managing multiple AI models running simultaneously without overloading the system.

Scalability measures how well inference performance improves when additional computational resources are allocated. In some cases, adding more GPUs or TPUs increases throughput significantly, but in other scenarios, bottlenecks such as memory bandwidth limitations or network latency may limit scaling efficiency. Benchmarks also assess how well a system balances multiple concurrent models in real-world deployment, where different AI-powered features may need to run at the same time without interference.

For cloud-based AI, benchmarks evaluate how efficiently a system handles fluctuating demand, ensuring that inference servers can dynamically allocate resources without compromising latency. In mobile and embedded AI, efficient multi-model execution is essential for running multiple AI-powered features simultaneously without degrading system performance.

12.7.2.7 Power Consumption & Energy Efficiency

Since inference workloads run continuously in production, power consumption and energy efficiency are critical considerations. This is particularly important for mobile and edge devices, where battery life and thermal constraints limit available computational resources. Even in large-scale cloud environments, power efficiency directly impacts operational costs and sustainability goals.

The energy required for a single inference is often measured in joules per inference, reflecting how efficiently a system processes inputs while minimizing power draw. In cloud-based inference, efficiency is commonly expressed as queries per second per watt (QPS/W) to quantify how well a system balances performance and energy consumption. For mobile AI applications, optimizing inference power consumption extends battery life and allows models to run efficiently on resource-constrained devices. Reducing energy use also plays

[7] Serverless AI: A deployment model where inference workloads are executed on demand, eliminating the need for dedicated compute resources but introducing cold-start latency challenges.

a key role in making large-scale AI systems more environmentally sustainable, ensuring that computational advancements align with energy-conscious deployment strategies. By balancing power consumption with performance, energy-efficient inference systems enable AI to scale sustainably across diverse applications, from data centers to edge devices.

12.7.3 Inference Performance Evaluation

Evaluating inference performance is a critical step in understanding how well machine learning systems meet the demands of real-world applications. Unlike training, which is typically conducted offline, inference systems must process inputs and generate predictions efficiently across a wide range of deployment scenarios. Metrics such as latency, throughput, memory usage, and energy efficiency provide a structured way to measure system performance and identify areas for improvement.

Table 12.3 below summarizes the key metrics used to evaluate inference systems, highlighting their relevance to different contexts. While each metric offers unique insights, it is important to approach inference benchmarking holistically. Trade-offs between metrics, including speed versus accuracy and throughput versus power consumption, are common, and understanding these trade-offs is essential for effective system design.

Table 12.3: Inference benchmark metrics and evaluation dimensions.

Category	Key Metrics	Example Benchmark Use
Latency and Tail Latency	Mean latency (ms/request); Tail latency (p95, p99, p99.9)	Evaluating real-time performance for safety-critical AI
Throughput and Efficiency	Queries per second (QPS); Frames per second (FPS); Batch throughput	Comparing large-scale cloud inference systems
Numerical Precision Impact	Accuracy degradation (FP32 vs. INT8); Speedup from reduced precision	Balancing accuracy vs. efficiency in optimized inference
Memory Footprint	Model size (MB/GB); RAM usage (MB); Memory bandwidth utilization	Assessing feasibility for edge and mobile deployments
Cold-Start and Load Time	Model load time (s); First inference latency (s)	Evaluating responsiveness in serverless AI
Scalability	Efficiency under load; Multi-model serving performance	Measuring robustness for dynamic, high-demand systems
Power and Energy Efficiency	Power consumption (Watts); Performance per Watt (QPS/W)	Optimizing energy use for mobile and sustainable AI

12.7.3.1 Inference Systems Considerations

Inference systems face unique challenges depending on where and how they are deployed. Real-time applications, such as self-driving cars or voice assistants, require low latency to ensure timely responses, while large-scale cloud deployments focus on maximizing throughput to handle millions of queries. Edge devices, on the other hand, are constrained by memory and power, making efficiency critical.

One of the most important aspects of evaluating inference performance is understanding the trade-offs between metrics. For example, optimizing for high throughput might increase latency, making a system unsuitable for real-time applications. Similarly, reducing numerical precision improves power efficiency and speed but may lead to minor accuracy degradation. A thoughtful evaluation must balance these trade-offs to align with the intended application.

The deployment environment also plays a significant role in determining evaluation priorities. Cloud-based systems often prioritize scalability and adaptability to dynamic workloads, while mobile and edge systems require careful attention to memory usage and energy efficiency. These differing priorities mean that benchmarks must be tailored to the context of the system's use, rather than relying on one-size-fits-all evaluations.

Ultimately, evaluating inference performance requires a holistic approach. Focusing on a single metric, such as latency or energy efficiency, provides an incomplete picture. Instead, all relevant dimensions must be considered together to ensure that the system meets its functional, resource, and performance goals in a balanced way.

12.7.3.2 Inference Benchmark Pitfalls

Even with well-defined metrics, benchmarking inference systems can be challenging. Missteps during the evaluation process often lead to misleading conclusions. Below are common pitfalls that students and practitioners should be aware of when analyzing inference performance.

Overemphasis on Average Latency. While average latency provides a baseline measure of response time, it fails to capture how a system performs under peak load. In real-world scenarios, worst-case latency, which is captured through metrics such as p95 or p99 tail latency, can significantly impact system reliability. For instance, a conversational AI system may fail to provide timely responses if occasional latency spikes exceed acceptable thresholds.

Ignoring Memory & Energy Constraints. A model with excellent throughput or latency may be unsuitable for mobile or edge deployments if it requires excessive memory or power. For example, an inference system designed for cloud environments might fail to operate efficiently on a battery-powered device. Proper benchmarks must consider memory footprint and energy consumption to ensure practicality across deployment contexts.

Ignoring Cold-Start Performance. In serverless environments, where models are loaded on demand, cold-start latency is a critical factor. Ignoring the time it takes to initialize a model and process the first request can result in unrealistic expectations for responsiveness. Evaluating both model load time and first-inference latency ensures that systems are designed to meet real-world responsiveness requirements.

Isolated Metrics Evaluation. Benchmarking inference systems often involves balancing competing metrics. For example, maximizing batch throughput might degrade latency, while aggressive quantization could reduce accuracy. Focusing on a single metric without considering its impact on others can lead to incomplete or misleading evaluations. Comprehensive benchmarks must account for these interactions.

Linear Scaling Assumption. Inference performance does not always scale proportionally with additional resources. Bottlenecks such as memory bandwidth, thermal limits, or communication overhead can limit the benefits of adding more GPUs or TPUs. Benchmarks that assume linear scaling behavior may overestimate system performance, particularly in distributed deployments.

Ignoring Application Requirements. Generic benchmarking results may fail to account for the specific needs of an application. For instance, a benchmark optimized for cloud inference might be irrelevant for edge devices, where energy and memory constraints dominate. Tailoring benchmarks to the deployment context ensures that results are meaningful and actionable.

12.7.3.3 Final Thoughts

Inference benchmarks are essential tools for understanding system performance, but their utility depends on careful and holistic evaluation. Metrics like latency, throughput, memory usage, and energy efficiency provide valuable insights, but their importance varies depending on the application and deployment context. Students should approach benchmarking as a process of balancing multiple priorities, rather than optimizing for a single metric.

Avoiding common pitfalls and considering the trade-offs between different metrics allows practitioners to design inference systems that are reliable, efficient, and suitable for real-world deployment. The ultimate goal of benchmarking is to guide system improvements that align with the demands of the intended application.

12.7.4 MLPerf Inference Benchmarks

The MLPerf Inference benchmark, developed by MLCommons, provides a standardized framework for evaluating machine learning inference performance across a range of deployment environments. Initially, MLPerf started with a single inference benchmark, but as machine learning systems expanded into diverse applications, it became clear that a one-size-fits-all benchmark was insufficient. Different inference scenarios, including cloud-based AI services and resource-constrained embedded devices, demanded tailored evaluations. This realization led to the development of a family of MLPerf inference benchmarks, each designed to assess performance within a specific deployment setting.

12.7.4.1 MLPerf Inference

MLPerf Inference serves as the baseline benchmark, originally designed to evaluate large-scale inference systems. It primarily focuses on data center and cloud-based inference workloads, where high throughput, low latency, and efficient resource utilization are essential. The benchmark assesses performance across a range of deep learning models, including image classification, object detection, natural language processing, and recommendation systems. This version of MLPerf remains the gold standard for comparing AI accelerators, GPUs, TPUs, and CPUs in high-performance computing environments.

12.7.4.2 MLPerf Mobile

MLPerf Mobile extends MLPerf's evaluation framework to smartphones and other mobile devices. Unlike cloud-based inference, mobile inference operates under strict power and memory constraints, requiring models to be optimized for efficiency without sacrificing responsiveness. The benchmark measures latency and responsiveness for real-time AI tasks, such as camera-based scene detection, speech recognition, and augmented reality applications. MLPerf Mobile has become an industry standard for assessing AI performance on flagship smartphones and mobile AI chips, helping developers optimize models for on-device AI workloads.

12.7.4.3 MLPerf Client

MLPerf Client focuses on inference performance on consumer computing devices, such as laptops, desktops, and workstations. This benchmark addresses local AI workloads that run directly on personal devices, eliminating reliance on cloud inference. Tasks such as real-time video editing, speech-to-text transcription, and AI-enhanced productivity applications fall under this category. Unlike cloud-based benchmarks, MLPerf Client evaluates how AI workloads interact with general-purpose hardware, such as CPUs, discrete GPUs, and integrated Neural Processing Units (NPUs), making it relevant for consumer and enterprise AI applications.

12.7.4.4 MLPerf Tiny

MLPerf Tiny was created to benchmark embedded and ultra-low-power AI systems, such as IoT devices, wearables, and microcontrollers. Unlike other MLPerf benchmarks, which assess performance on powerful accelerators, MLPerf Tiny evaluates inference on devices with limited compute, memory, and power resources. This benchmark is particularly relevant for applications such as smart sensors, AI-driven automation, and real-time industrial monitoring, where models must run efficiently on hardware with minimal processing capabilities. MLPerf Tiny plays a crucial role in the advancement of AI at the edge, helping developers optimize models for constrained environments.

12.7.4.5 Continued Expansion

The evolution of MLPerf Inference from a single benchmark to a spectrum of benchmarks reflects the diversity of AI deployment scenarios. Different environments, including cloud, mobile, desktop, and embedded environments, have unique constraints and requirements, and MLPerf provides a structured way to evaluate AI models accordingly.

MLPerf is an essential tool for:

- Understanding how inference performance varies across deployment settings.
- Learning which performance metrics are most relevant for different AI applications.
- Optimizing models and hardware choices based on real-world usage constraints.

Recognizing the necessity of tailored inference benchmarks deepens our understanding of AI deployment challenges and highlights the importance of benchmarking in developing efficient, scalable, and practical machine learning systems.

12.8 Energy Efficiency Measurement

As machine learning expands into diverse applications, concerns about its growing power consumption and ecological footprint have intensified. While performance benchmarks help optimize speed and accuracy, they do not always account for energy efficiency, which is an increasingly critical factor in real-world deployment. Efficient inference is particularly important in scenarios where power is a limited resource, such as mobile devices, embedded AI, and cloud-scale inference workloads. The need to optimize both performance and power consumption has led to the development of standardized energy efficiency benchmarks.

However, measuring power consumption in machine learning systems presents unique challenges. The energy demands of ML models vary dramatically across deployment environments, as shown in Table 12.4. This wide spectrum, spanning from TinyML devices consuming mere microwatts to data center racks requiring kilowatts, illustrates the fundamental challenge in creating standardized benchmarking methodologies (Henderson et al. 2020a).

Table 12.4: Power consumption across ML deployment scales

Category	Device Type	Power Consumption
Tiny	Neural Decision Processor (NDP)	150 µW
Tiny	M7 Microcontroller	25 mW
Mobile	Raspberry Pi 4	3.5 W
Mobile	Smartphone	4 W
Edge	Smart Camera	10-15 W
Edge	Edge Server	65-95 W
Cloud	ML Server Node	300-500 W
Cloud	ML Server Rack	4-10 kW

This dramatic range in power requirements, which spans over four orders of magnitude, presents significant challenges for measurement and benchmarking. Creating a unified methodology requires careful consideration of each scale's unique characteristics. For example, accurately measuring microwatt-level consumption in TinyML devices demands different instrumentation and techniques than monitoring kilowatt-scale server racks. Any comprehensive benchmarking framework must accommodate these vastly different scales while ensuring measurements remain consistent, fair, and reproducible across diverse hardware configurations.

12.8.1 Power Measurement Boundaries

Figure 12.8 illustrates how power consumption is measured at different system scales, from TinyML devices to full-scale data center inference nodes. Each scenario highlights distinct measurement boundaries, shown in green, which indicate the components included in energy accounting. Components outside these boundaries, shown with red dashed outlines, are excluded from power measurements.

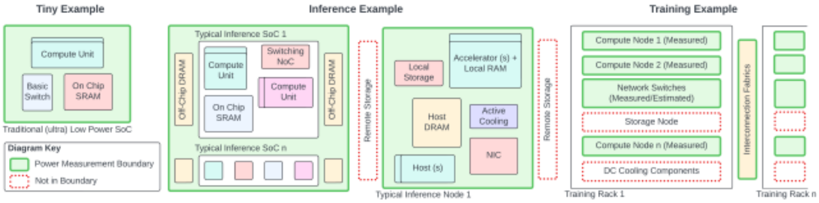


Figure 12.8: MLPerf Power system measurement diagram. Source: Tschand et al. (2024).

The diagram is organized into three categories, Tiny, Inference, and Training examples, each reflecting different measurement scopes based on system architecture and deployment environment. In TinyML systems, the entire low-power SoC, including compute, memory, and basic interconnects, typically falls within the measurement boundary. Inference nodes introduce more complexity, incorporating multiple SoCs, local storage, accelerators, and memory, while often excluding remote storage and off-chip components. Training deployments span multiple racks, where only selected elements, including compute nodes and network switches, are measured, while storage systems, cooling infrastructure, and parts of the interconnect fabric are often excluded.

System-level power measurement offers a more holistic view than measuring individual components in isolation. While component-level metrics (e.g., accelerator or processor power) are valuable for performance tuning, real-world ML workloads involve intricate interactions between compute units, memory systems, and supporting infrastructure. For instance, memory-bound inference tasks can consume up to 60% of total system power on data movement alone.

Shared infrastructure presents additional challenges. In data centers, resources such as cooling systems and power delivery are shared across workloads, complicating attribution of energy use to specific ML tasks. Cooling alone can account for 20-30% of total facility power consumption, making it a major factor in energy efficiency assessments (Barroso, Clidaras, and Hölzle 2013). Even at the edge, components like memory and I/O interfaces may serve both ML and non-ML functions, further blurring measurement boundaries.

Modern hardware also introduces variability through dynamic power management. Features like dynamic voltage and frequency scaling (DVFS) can cause power consumption to vary by 30-50% for the same ML model, depending on system load and concurrent activity.

Finally, support infrastructure, with a particular emphasis on cooling, has a significant impact on total energy use in large-scale deployments. Data centers must maintain operational temperatures, typically between 20-25°C, to ensure system reliability. Cooling overhead is captured in the Power Usage Effectiveness (PUE) metric, which ranges from 1.1 in highly efficient facilities to over 2.0 in less optimized ones (Barroso, Hölzle, and Ranganathan 2019). Even edge devices require basic thermal management, with cooling accounting for 5-10% of overall power consumption.

12.8.2 Performance vs Energy Efficiency

A critical consideration in ML system design is the relationship between performance and energy efficiency. Maximizing raw performance often leads to diminishing returns in energy efficiency. For example, increasing processor frequency by 20% might yield only a 5% performance improvement while increasing power consumption by 50%. This non-linear relationship means that the most energy-efficient operating point is often not the highest performing one.

In many deployment scenarios, particularly in battery-powered devices, finding the optimal balance between performance and energy efficiency is crucial. For instance, reducing model precision from FP32 to INT8 might reduce accuracy by 1-2% but can improve energy efficiency by 3-4x. Similarly, batch processing can improve throughput efficiency at the cost of increased latency.

These tradeoffs span three key dimensions: accuracy, performance, and energy efficiency. Model quantization illustrates this relationship clearly, reducing numerical precision from FP32 to INT8 typically results in a small accuracy drop (1-2%), but it can improve both inference speed and energy efficiency by 3-4x. Similarly, techniques like pruning and model compression require carefully balancing accuracy losses against efficiency gains. Finding the optimal operating point among these three factors depends heavily on deployment requirements; mobile applications might prioritize energy efficiency, while cloud services might optimize for accuracy at the cost of higher power consumption.

As benchmarking methodologies continue to evolve, energy efficiency metrics will play an increasingly central role in AI optimization. Future advancements in sustainable AI benchmarking[8] will help researchers and engineers design systems that balance performance, power consumption, and environmental impact, ensuring that ML systems operate efficiently without unnecessary energy waste.

12.8.3 Standardized Power Measurement

While power measurement techniques, such as SPEC Power, have long existed for general computing systems (Lange 2009), machine learning workloads present unique challenges that require specialized measurement approaches. Machine learnign systems exhibit distinct power consumption patterns characterized by phases of intense computation interspersed with data movement and preprocessing operations. These patterns vary significantly across different types of models and tasks. A large language model's power profile looks very different from that of a computer vision inference task.

Direct power measurement requires careful consideration of sampling rates and measurement windows. For example, transformer model inference creates short, intense power spikes during attention computations, requiring high-frequency sampling (> 1 KHz) to capture accurately. In contrast, CNN inference tends to show more consistent power draw patterns that can be captured with lower sampling rates. The measurement duration must also account for ML-specific behaviors like warm-up periods, where initial inferences may consume more power due to cache population and pipeline initialization.

[8] Reducing the environmental impact of machine learning by improving energy efficiency, using renewable energy sources, and designing models that require fewer computational resources.

Memory access patterns in ML workloads significantly impact power consumption measurements. While traditional compute benchmarks might focus primarily on processor power, ML systems often spend substantial energy moving data between memory hierarchies. For example, recommendation models like DLRM can spend more energy on memory access than computation. This requires measurement approaches that can capture both compute and memory subsystem power consumption.

Accelerator-specific considerations further complicate power measurement. Many ML systems employ specialized hardware like GPUs, TPUs, or NPUs. These accelerators often have their own power management schemes and can operate independently of the main system processor. Accurate measurement requires capturing power consumption across all relevant compute units while maintaining proper time synchronization. This is particularly challenging in heterogeneous systems that may dynamically switch between different compute resources based on workload characteristics or power constraints.

The scale and distribution of ML workloads also influences measurement methodology. In distributed training scenarios, power measurement must account for both local compute power and the energy cost of gradient synchronization across nodes. Similarly, edge ML deployments must consider both active inference power and the energy cost of model updates or data preprocessing.

Batch size and throughput considerations add another layer of complexity. Unlike traditional computing workloads, ML systems often process inputs in batches to improve computational efficiency. However, the relationship between batch size and power consumption is non-linear. While larger batches generally improve compute efficiency, they also increase memory pressure and peak power requirements. Measurement methodologies must therefore capture power consumption across different batch sizes to provide a complete efficiency profile.

System idle states require special attention in ML workloads, particularly in edge scenarios where systems operate intermittently, actively processing when new data arrives, then entering low-power states between inferences. A wake-word detection Tiny ML system, for instance, might only actively process audio for a small fraction of its operating time, making idle power consumption a critical factor in overall efficiency.

Temperature effects play a crucial role in ML system power measurement. Sustained ML workloads can cause significant temperature increases, triggering thermal throttling and changing power consumption patterns. This is especially relevant in edge devices where thermal constraints may limit sustained performance. Measurement methodologies must account for these thermal effects and their impact on power consumption, particularly during extended benchmarking runs.

12.8.4 MLPerf Power Case Study

MLPerf Power (Tschand et al. 2024) is a standard methodolgy for measuring energy efficiency in machine learning systems. This comprehensive benchmarking framework provides accurate assessment of power consumption across diverse ML deployments. At the datacenter level, it measures power usage in large-scale AI workloads, where energy consumption optimization directly impacts operational costs. For edge computing, it evaluates power efficiency in consumer devices like smartphones and laptops, where battery life constraints are paramount. In tiny inference scenarios, it assesses energy consumption for ultra-low-power AI systems, particularly IoT sensors and microcontrollers operating with strict power budgets.

The MLPerf Power methodology relies on standardized measurement protocols that adapt to various hardware architectures, ranging from general-purpose CPUs to specialized AI accelerators. This standardization ensures meaningful cross-platform comparisons while maintaining measurement integrity across different computing scales.

The benchmark has accumulated thousands of reproducible measurements submitted by industry organizations, which demonstrates their latest hardware capabilities and the sector-wide focus on energy-efficient AI technology. Figure 12.9 illustrates the evolution of energy efficiency across system scales through successive MLPerf versions.

The MLPerf Power methodology adapts to different hardware architectures, ranging from general-purpose CPUs to specialized AI accelerators, while maintaining a uniform measurement standard. This ensures that comparisons across platforms are meaningful and unbiased.

Across the versions and ML deployment scales of the MLPerf benchmark suite, industry organizations have submitted reproducible measurements on their most recent hardware to observe and quantify the industry-wide emphasis on optimizing AI technology for energy efficiency. Figure 12.9 shows the trends in energy efficiency from tiny to datacenter scale systems across MLPerf versions.

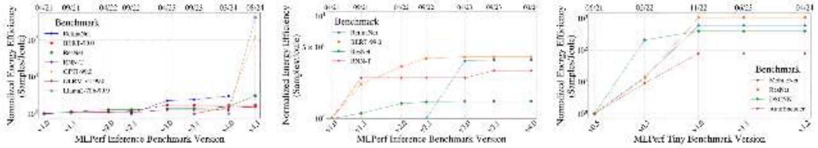


Figure 12.9: Comparison of energy efficiency trends for MLPerf Power datacenter, edge, and tiny inference submissions across versions. Source: Tschand et al. (2024).

Analysis of these trends reveals two significant patterns: first, a plateauing of energy efficiency improvements across all three scales for traditional ML workloads, and second, a dramatic increase in energy efficiency specifically for generative AI applications. This dichotomy suggests both the maturation of optimization techniques for conventional ML tasks and the rapid innovation occurring in the generative AI space. These trends underscore the dual challenges facing the field: developing novel approaches to break through efficiency plateaus while ensuring sustainable scaling practices for increasingly powerful generative AI models.

12.9 Challenges & Limitations

Benchmarking provides a structured framework for evaluating the performance of AI systems, but it comes with significant challenges. If these challenges are not properly addressed, they can undermine the credibility and usefulness of benchmarking results. One of the most fundamental issues is incomplete problem coverage. Many benchmarks, while useful for controlled comparisons, fail to capture the full diversity of real-world applications. For instance, common image classification datasets, such as CIFAR-10, contain a limited variety of images. As a result, models that perform well on these datasets may struggle when applied to more complex, real-world scenarios with greater variability in lighting, perspective, and object composition.

Another challenge is statistical insignificance, which arises when benchmark evaluations are conducted on too few data samples or trials. For example, testing an optical character recognition (OCR) system on a small dataset may not accurately reflect its performance on large-scale, noisy text documents. Without sufficient trials and diverse input distributions, benchmarking results may be misleading or fail to capture true system reliability.

Reproducibility is also a major concern. Benchmark results can vary significantly depending on factors such as hardware configurations, software versions, and system dependencies. Small differences in compilers, numerical precision, or library updates can lead to inconsistent performance measurements across different environments. To mitigate this issue, MLPerf addresses reproducibility by providing reference implementations, standardized test environments, and strict submission guidelines. Even with these efforts, achieving true consistency across diverse hardware platforms remains an ongoing challenge.

A more fundamental limitation of benchmarking is the risk of misalignment with real-world goals. Many benchmarks emphasize metrics such as speed, accuracy, and throughput, but practical AI deployments often require balancing multiple objectives, including power efficiency, cost, and robustness. A model that achieves state-of-the-art accuracy on a benchmark may be impractical for deployment if it consumes excessive energy or requires expensive hardware. Furthermore, benchmarks can quickly become outdated due to the rapid evolution of AI models and hardware. New techniques may emerge that render existing benchmarks less relevant, necessitating continuous updates to keep benchmarking methodologies aligned with state-of-the-art developments.

While these challenges affect all benchmarking efforts, the most pressing concern is the role of benchmark engineering, which introduces the risk of over-optimization for specific benchmark tasks rather than meaningful improvements in real-world performance.

12.9.1 Environmental Conditions

Environmental conditions in AI benchmarking refer to the physical and operational circumstances under which experiments are conducted. These conditions, while often overlooked, can significantly influence benchmark results and impact the reproducibility of experiments. Physical environmental factors include ambient temperature, humidity, air quality, and altitude. These elements can affect hardware performance in subtle but measurable ways. For instance, elevated temperatures may lead to thermal throttling[9] in processors, potentially reducing computational speed and affecting benchmark outcomes.

[9] Thermal Throttling: A mechanism in computer processors that reduces performance to prevent overheating, often triggered by excessive computational load or inadequate cooling.

Similarly, variations in altitude can impact cooling system efficiency and hard drive performance due to changes in air pressure.

Operational environmental factors encompass the broader system context in which benchmarks are executed. This includes background processes running on the system, network conditions, and power supply stability. The presence of other active programs or services can compete for computational resources, potentially altering the performance characteristics of the model under evaluation. To ensure the validity and reproducibility of benchmark results, it is essential to document and control these environmental conditions to the extent possible. This may involve conducting experiments in temperature-controlled environments, monitoring and reporting ambient conditions, standardizing the operational state of benchmark systems, and documenting any background processes or system loads.

In scenarios where controlling all environmental variables is impractical, such as in distributed or cloud-based benchmarking, it becomes essential to report these conditions in detail. This information allows other researchers to account for potential variations when interpreting or attempting to reproduce results. As machine learning models are increasingly deployed in diverse real-world environments, understanding the impact of environmental conditions on model performance becomes even more critical. This knowledge not only ensures more accurate benchmarking but also informs the development of robust models capable of consistent performance across varying operational conditions.

12.9.2 Hardware Lottery

A critical issue in benchmarking is what has been described as the hardware lottery, a concept introduced by (Ahmed et al. 2021). The success of a machine learning model is often dictated not only by its architecture and training data but also by how well it aligns with the underlying hardware used for inference. Some models perform exceptionally well, not because they are inherently better, but because they are optimized for the parallel processing capabilities of GPUs or TPUs. Meanwhile, other promising architectures may be overlooked because they do not map efficiently to dominant hardware platforms.

This dependence on hardware compatibility introduces biases into benchmarking. A model that is highly efficient on a specific GPU may perform poorly on a CPU or a custom AI accelerator. For instance, Figure 12.10 compares the performance of models across different hardware platforms. The multi-hardware models show comparable results to "MobileNetV3 Large min" on both the CPU `uint8` and GPU configurations. However, these multi-hardware models demonstrate significant performance improvements over the MobileNetV3 Large baseline when run on the EdgeTPU and DSP hardware. This emphasizes the variable efficiency of multi-hardware models in specialized computing environments.

Without careful benchmarking across diverse hardware configurations, the field risks favoring architectures that "win" the hardware lottery rather than selecting models based on their intrinsic strengths. This bias can shape research directions, influence funding allocation, and impact the design of next-generation

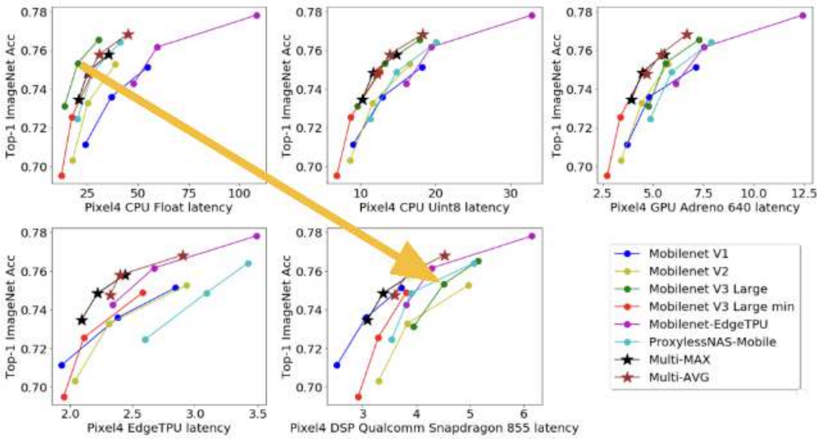


Figure 12.10: Accuracy-latency trade-offs of multiple ML models and how they perform on various hardware. Source: Chu et al. (2021).

AI systems. In extreme cases, it may even stifle innovation by discouraging exploration of alternative architectures that do not align with current hardware trends.

12.9.3 Benchmark Engineering

While the hardware lottery is an unintended consequence of hardware trends, benchmark engineering is an intentional practice where models or systems are explicitly optimized to excel on specific benchmark tests. This practice can lead to misleading performance claims and results that do not generalize beyond the benchmarking environment.

Benchmark engineering occurs when AI developers fine-tune hyperparameters, preprocessing techniques, or model architectures specifically to maximize benchmark scores rather than improve real-world performance. For example, an object detection model might be carefully optimized to achieve record-low latency on a benchmark but fail when deployed in dynamic, real-world environments with varying lighting, motion blur, and occlusions. Similarly, a language model might be tuned to excel on benchmark datasets but struggle when processing conversational speech with informal phrasing and code-switching.

The pressure to achieve high benchmark scores is often driven by competition, marketing, and research recognition. Benchmarks are frequently used to rank AI models and systems, creating an incentive to optimize specifically for them. While this can drive technical advancements, it also risks prioritizing benchmark-specific optimizations at the expense of broader generalization.

12.9.4 Bias & Over-Optimization

To ensure that benchmarks remain useful and fair, several strategies can be employed. Transparency is one of the most important factors in maintaining benchmarking integrity. Benchmark submissions should include detailed documentation on any optimizations applied, ensuring that improvements are clearly distinguished from benchmark-specific tuning. Researchers and developers should report both benchmark performance and real-world deployment results to provide a complete picture of a system's capabilities.

Another approach is to diversify and evolve benchmarking methodologies. Instead of relying on a single static benchmark, AI systems should be evaluated across multiple, continuously updated benchmarks that reflect real-world complexity. This reduces the risk of models being overfitted to a single test set and encourages general-purpose improvements rather than narrow optimizations.

Standardization and third-party verification can also help mitigate bias. By establishing industry-wide benchmarking standards and requiring independent third-party audits of results, the AI community can improve the reliability and credibility of benchmarking outcomes. Third-party verification ensures that reported results are reproducible across different settings and helps prevent unintentional benchmark gaming.

Another important strategy is application-specific testing. While benchmarks provide controlled evaluations, real-world deployment testing remains essential. AI models should be assessed not only on benchmark datasets but also in practical deployment environments. For instance, an autonomous driving model should be tested in a variety of weather conditions and urban settings rather than being judged solely on controlled benchmark datasets.

Finally, fairness across hardware platforms must be considered. Benchmarks should test AI models on multiple hardware configurations to ensure that performance is not being driven solely by compatibility with a specific platform. This helps reduce the risk of the hardware lottery and provides a more balanced evaluation of AI system efficiency.

12.9.5 Benchmark Evolution

One of the greatest challenges in benchmarking is that benchmarks are never static. As AI systems evolve, so must the benchmarks that evaluate them. What defines "good performance" today may be irrelevant tomorrow as models, hardware, and application requirements change. While benchmarks are essential for tracking progress, they can also quickly become outdated, leading to over-optimization for old metrics rather than real-world performance improvements.

This evolution is evident in the history of AI benchmarks. Early model benchmarks, for instance, focused heavily on image classification and object detection, as these were some of the first widely studied deep learning tasks. However, as AI expanded into natural language processing, recommendation systems, and generative AI, it became clear that these early benchmarks no longer reflected the most important challenges in the field. In response, new benchmarks emerged to measure language understanding (A. Wang et al. 2018, 2019) and generative AI (Liang et al. 2022).

Benchmark evolution extends beyond the addition of new tasks to encompass new dimensions of performance measurement. While traditional AI benchmarks emphasized accuracy and throughput, modern applications demand evaluation across multiple criteria: fairness, robustness, scalability, and energy efficiency. Figure 12.11 illustrates this complexity through scientific applications, which span orders of magnitude in their performance requirements. For instance, Large Hadron Collider sensors must process data at rates approaching 10^{14} bytes per second with nanosecond-scale computation times, while mobile applications operate at 10^4 bytes per second with longer computational windows. This range of requirements necessitates specialized benchmarks—for example, edge AI applications require benchmarks like MLPerf that specifically evaluate performance under resource constraints and scientific application domains need their own "Fast ML for Science" benchmarks (Duarte et al. 2022a).

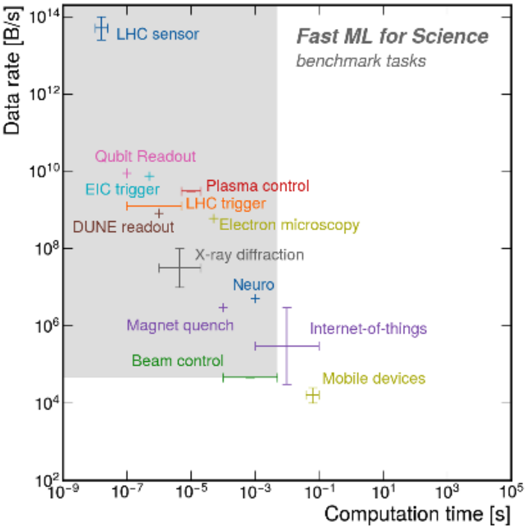


Figure 12.11: Data rate and computation time requirements of emerging scientific applications. Source: (Duarte et al. 2022b).

The need for evolving benchmarks also presents a challenge: stability versus adaptability. On the one hand, benchmarks must remain stable for long enough to allow meaningful comparisons over time. If benchmarks change too frequently, it becomes difficult to track long-term progress and compare new results with historical performance. On the other hand, failing to update benchmarks leads to stagnation, where models are optimized for outdated tasks rather than advancing the field. Striking the right balance between benchmark longevity and adaptation is an ongoing challenge for the AI community.

Despite these difficulties, evolving benchmarks is essential for ensuring that AI progress remains meaningful. Without updates, benchmarks risk becoming detached from real-world needs, leading researchers and engineers to focus on optimizing models for artificial test cases rather than solving practical challenges. As AI continues to expand into new domains, benchmarking must keep pace, ensuring that performance evaluations remain relevant, fair, and aligned with real-world deployment scenarios.

12.9.6 MLPerf's Role

MLPerf has played a crucial role in improving benchmarking by reducing bias, increasing generalizability, and ensuring benchmarks evolve alongside AI advancements. One of its key contributions is the standardization of benchmarking environments. By providing reference implementations, clearly defined rules, and reproducible test environments, MLPerf ensures that performance results are consistent across different hardware and software platforms, reducing variability in benchmarking outcomes.

Recognizing that AI is deployed in a variety of real-world settings, MLPerf has also introduced different categories of inference benchmarks. The inclusion of MLPerf Inference, MLPerf Mobile, MLPerf Client, and MLPerf Tiny reflects an effort to evaluate models in the contexts where they will actually be deployed. This approach mitigates issues such as the hardware lottery by ensuring that AI systems are tested across diverse computational environments, rather than being over-optimized for a single hardware type.

Beyond providing a structured benchmarking framework, MLPerf is continuously evolving to keep pace with the rapid progress in AI. New tasks are incorporated into benchmarks to reflect emerging challenges, such as generative AI models and energy-efficient computing, ensuring that evaluations remain relevant and forward-looking. By regularly updating its benchmarking methodologies, MLPerf helps prevent benchmarks from becoming outdated or encouraging overfitting to legacy performance metrics.

By prioritizing fairness, transparency, and adaptability, MLPerf ensures that benchmarking remains a meaningful tool for guiding AI research and deployment. Instead of simply measuring raw speed or accuracy, MLPerf's evolving benchmarks aim to capture the complexities of real-world AI performance, ultimately fostering more reliable, efficient, and impactful AI systems.

12.10 Beyond System Benchmarking

While this chapter has primarily focused on system benchmarking, AI performance is not determined by system efficiency alone. Machine learning models and datasets play an equally crucial role in shaping AI capabilities. Model benchmarking evaluates algorithmic performance, while data benchmarking ensures that training datasets are high-quality, unbiased, and representative of real-world distributions. Understanding these aspects is vital because AI systems are not just computational pipelines—they are deeply dependent on the models they execute and the data they are trained on.

12.10.1 Model Benchmarking

Model benchmarks measure how well different machine learning algorithms perform on specific tasks. Historically, benchmarks focused almost exclusively on accuracy, but as models have grown more complex, additional factors, including fairness, robustness, efficiency, and generalizability, have become equally important.

The evolution of machine learning has been largely driven by benchmark datasets. The MNIST dataset (Lecun et al. 1998) was one of the earliest catalysts, advancing handwritten digit recognition, while the ImageNet dataset (J. Deng et al. 2009) sparked the deep learning revolution in image classification. More recently, datasets like COCO (T.-Y. Lin et al. 2014) for object detection and

GPT-3's training corpus (T. B. Brown, Mann, Ryder, Subbiah, Kaplan, and al. 2020) have pushed the boundaries of model capabilities even further.

However, model benchmarks face significant limitations, particularly in the era of Large Language Models (LLMs). Beyond the traditional challenge of models failing in real-world conditions, commonly referred to as the Sim2Real gap, a new form of benchmark optimization has emerged, analogous to but distinct from classical benchmark engineering in computer systems. In traditional systems evaluation, developers would explicitly optimize their code implementations to perform well on benchmark suites like SPEC or TPC, which we discussed earlier under "Benchmark Engineering". In the case of LLMs, this phenomenon manifests through data rather than code: benchmark datasets may become embedded in training data, either inadvertently through web-scale training or deliberately through dataset curation (R. Xu et al. 2024). This creates fundamental challenges for model evaluation, as high performance on benchmark tasks may reflect memorization rather than genuine capability. The key distinction lies in the mechanism: while systems benchmark engineering occurred through explicit code optimization, LLM benchmark adaptation can occur implicitly through data exposure during pre-training, raising new questions about the validity of current evaluation methodologies.

These challenges extend beyond just LLMs. Traditional machine learning systems continue to struggle with problems of overfitting and bias. The Gender Shades project (Buolamwini and Gebru 2018), for instance, revealed that commercial facial recognition models performed significantly worse on darker-skinned individuals, highlighting the critical importance of fairness in model evaluation. Such findings underscore the limitations of focusing solely on aggregate accuracy metrics.

Moving forward, we must fundamentally rethink its approach to benchmarking. This evolution requires developing evaluation frameworks that go beyond traditional metrics to assess multiple dimensions of model behavior—from generalization and robustness to fairness and efficiency. Key challenges include creating benchmarks that remain relevant as models advance, developing methodologies that can differentiate between genuine capabilities and artificial performance gains, and establishing standards for benchmark documentation and transparency. Success in these areas will help ensure that benchmark results provide meaningful insights about model capabilities rather than reflecting artifacts of training procedures or evaluation design.

12.10.2 Data Benchmarking

The evolution of artificial intelligence has traditionally focused on model-centric approaches, emphasizing architectural improvements and optimization techniques. However, contemporary AI development reveals that data quality, rather than model design alone, often determines performance boundaries. This recognition has elevated data benchmarking to a critical field that ensures AI models learn from datasets that are high-quality, diverse, and free from bias.

This evolution represents a fundamental shift from model-centric to data-centric AI approaches, as illustrated in Figure 12.12. The traditional model-centric paradigm focuses on enhancing model architectures, refining algorithms, and improving computational efficiency while treating datasets as fixed components. In contrast, the emerging data-centric approach systematically improves dataset quality through better annotations, increased diversity, and bias reduction, while maintaining consistent model architectures and system configurations. Research increasingly demonstrates that methodical dataset enhancement can yield superior performance gains compared to model refinements alone, challenging the conventional emphasis on architectural innovation.

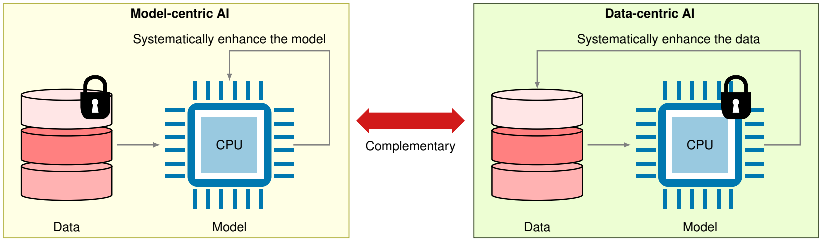
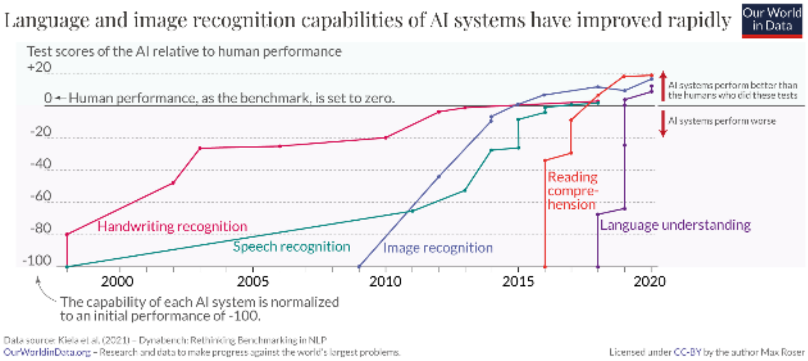


Figure 12.12: Comparison of model-centric and data-centric AI approaches. Model-centric AI focuses on improving architectures, while data-centric AI emphasizes enhancing dataset quality. Both approaches are complementary in optimizing AI performance.

Data quality's primacy in AI development reflects a fundamental shift in understanding: superior datasets, not just sophisticated models, produce more reliable and robust AI systems. Initiatives like DataPerf and DataComp have emerged to systematically evaluate how dataset improvements affect model performance. For instance, DataComp (Nishigaki 2024) demonstrated that models trained on a carefully curated 30% subset of data achieved better results than those trained on the complete dataset, challenging the assumption that more data automatically leads to better performance (Northcutt, Athalye, and Mueller 2021).

A significant challenge in data benchmarking emerges from dataset saturation. When models achieve near-perfect accuracy on benchmarks like ImageNet, it becomes crucial to distinguish whether performance gains represent genuine advances in AI capability or merely optimization to existing test sets. Figure 12.13 illustrates this trend, showing AI systems surpassing human performance across various applications over the past decade.



Data source: Kiela et al. (2021) – Dynabench: Rethinking Benchmarking in NLP
OurWorldinData.org – Research and data to make progress against the world's largest problems.
Licensed under CC-BY by the author Max Roser

Figure 12.13: AI vs human performance. Source: Kiela et al. (2021)

This saturation phenomenon raises fundamental methodological questions (Kiela et al. 2021). The MNIST dataset provides an illustrative example: certain test images, though nearly illegible to humans, were assigned specific labels during the dataset's creation in 1994. When models correctly predict these labels, their apparent superhuman performance may actually reflect memorization of dataset artifacts rather than true digit recognition capabilities.

These challenges extend beyond individual domains. The provocative question "Are we done with ImageNet?" (Beyer et al. 2020) highlights broader concerns about the limitations of static benchmarks. Models optimized for fixed datasets often struggle with distribution shifts—real-world changes that occur after training data collection. This limitation has driven the development of dynamic benchmarking approaches, such as Dynabench (Kiela et al. 2021), which continuously evolves test data based on model performance to maintain benchmark relevance.

Current data benchmarking efforts encompass several critical dimensions. Label quality assessment remains a central focus, as explored in DataPerf's debugging challenge. Initiatives like MSWC (Mazumder et al. 2021) for speech recognition address bias and representation in datasets. Out-of-distribution generalization receives particular attention through benchmarks like RxRx and WILDS (Koh et al. 2021). These diverse efforts reflect a growing recognition that advancing AI capabilities requires not just better models and systems, but fundamentally better approaches to data quality assessment and benchmark design.

12.10.3 Benchmarking Trifecta

AI benchmarking has traditionally evaluated systems, models, and data as separate entities. However, real-world AI performance emerges from the interplay between these three components. A fast system cannot compensate for a poorly trained model, and even the most powerful model is constrained by the quality of the data it learns from. This interdependence necessitates a holistic benchmarking approach that considers all three dimensions together.

As illustrated in Figure 12.14, the future of benchmarking lies in an integrated framework that jointly evaluates system efficiency, model performance, and data quality. This approach enables researchers to identify optimization opportunities that remain invisible when these components are analyzed in isolation. For example, co-designing efficient AI models with hardware-aware optimizations and carefully curated datasets can lead to superior performance while reducing computational costs.

As AI continues to evolve, benchmarking methodologies must advance in tandem. Evaluating AI performance through the lens of systems, models, and data ensures that benchmarks drive improvements not just in accuracy, but also in efficiency, fairness, and robustness. This holistic perspective will be critical for developing AI that is not only powerful but also practical, scalable, and ethical.

12.11 Conclusion

"What gets measured gets improved." Benchmarking plays a foundational role in the advancement of AI, providing the essential measurements needed to

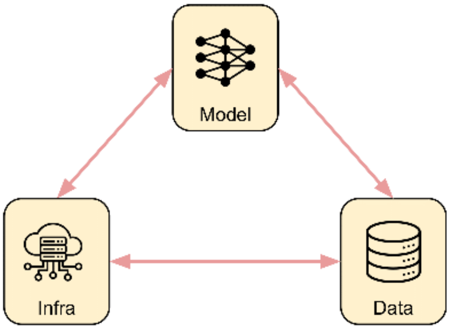


Figure 12.14: Benchmarking trifecta.

track progress, identify limitations, and drive innovation. This chapter has explored the multifaceted nature of benchmarking, spanning systems, models, and data, and has highlighted its critical role in optimizing AI performance across different dimensions.

ML system benchmarks enable optimizations in speed, efficiency, and scalability, ensuring that hardware and infrastructure can support increasingly complex AI workloads. Model benchmarks provide standardized tasks and evaluation metrics beyond accuracy, driving progress in algorithmic innovation. Data benchmarks, meanwhile, reveal key issues related to data quality, bias, and representation, ensuring that AI models are built on fair and diverse datasets.

While these components, systems, models, and data, are often evaluated in isolation, future benchmarking efforts will likely adopt a more integrated approach. By measuring the interplay between system, model, and data benchmarks, AI researchers and engineers can uncover new insights into the co-design of data, algorithms, and infrastructure. This holistic perspective will be essential as AI applications grow more sophisticated and are deployed across increasingly diverse environments.

Benchmarking is not static—it must continuously evolve to capture new AI capabilities, address emerging challenges, and refine evaluation methodologies. As AI systems become more complex and influential, the need for rigorous, transparent, and socially beneficial benchmarking standards becomes even more pressing. Achieving this requires close collaboration between industry, academia, and standardization bodies to ensure that benchmarks remain relevant, unbiased, and aligned with real-world needs.

Ultimately, benchmarking serves as the compass that guides AI progress. By persistently measuring and openly sharing results, we can navigate toward AI systems that are performant, robust, and trustworthy. However, benchmarking must also be aligned with human-centered principles, ensuring that AI serves society in a fair and ethical manner. The future of benchmarking is already expanding into new frontiers, including the evaluation of AI safety, fairness, and generative AI models, which will shape the next generation of AI benchmarks. These topics, while beyond the scope of this chapter, will be explored further in the discussion on Generative AI.

For those interested in emerging trends in AI benchmarking, the article *The Olympics of AI: Benchmarking Machine Learning Systems* provides a broader look at benchmarking efforts in robotics, extended reality, and neuromorphic computing. As benchmarking continues to evolve, it remains an essential tool for understanding, improving, and shaping the future of AI.

12.12 Resources

ℹ Slides

- Why is benchmarking important?
- Embedded inference benchmarking.

! Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 13

ML Operations

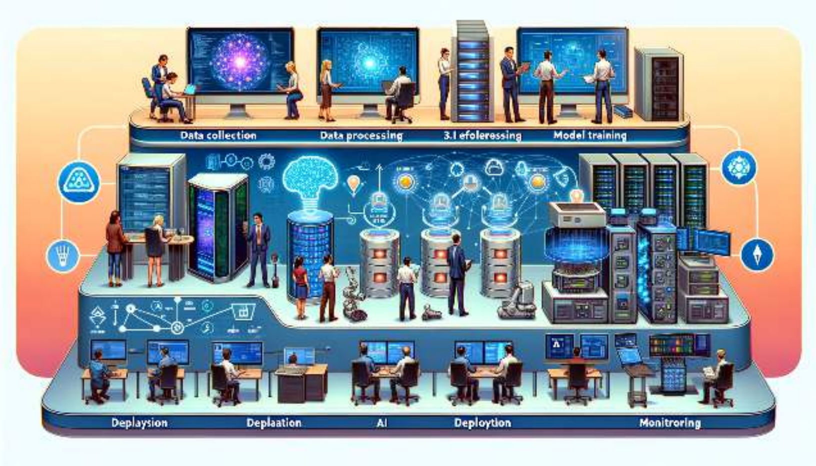


Figure 13.1: *DALL·E 3 Prompt: Create a detailed, wide rectangular illustration of an AI workflow. The image should showcase the process across six stages, with a flow from left to right: 1. Data collection, with diverse individuals of different genders and descents using a variety of devices like laptops, smartphones, and sensors to gather data. 2. Data processing, displaying a data center with active servers and databases with glowing lights. 3. Model training, represented by a computer screen with code, neural network diagrams, and progress indicators. 4. Model evaluation, featuring people examining data analytics on large monitors. 5. Deployment, where the AI is integrated into robotics, mobile apps, and industrial equipment. 6. Monitoring, showing professionals tracking AI performance metrics on dashboards to check for accuracy and concept drift over time. Each stage should be distinctly marked and the style should be clean, sleek, and modern with a dynamic and informative color scheme.*

Purpose

How do we operationalize machine learning principles in practice, and enable the continuous evolution of machine learning systems in production?

Developing machine learning systems does not end with training a performant model. As models are integrated into real-world applications, new demands arise around reliability, continuity, governance, and iteration. Operationalizing machine learning requires principles that help us understand how systems behave over time, including how data shifts, models degrade, and organizational processes adapt. In this context, several foundational questions emerge: How do we manage evolving data distributions? What infrastructure enables continuous delivery and real-time monitoring? How do we coordinate efforts across technical and organizational boundaries? What processes ensure reliability, reproducibility, and compliance under real-world constraints? These concerns are not peripheral, as they are central to constructing sustainable machine learning systems. Addressing them calls for a synthesis of software engineering, systems thinking, and organizational alignment. This shift (from building isolated models to engineering adaptive systems) marks a necessary evolution in how machine learning is developed, deployed, and maintained in practice.

Learning Objectives

- Define MLOps and explain its purpose in the machine learning lifecycle.
- Describe the key components of an MLOps pipeline.
- Discuss the significance of monitoring and observability in MLOps.
- Identify and describe the unique forms of technical debt that arise in ML systems.
- Describe the roles and responsibilities of key personnel involved in MLOps.
- Analyze the impact of operational maturity on ML system design and organizational structure.

13.1 Overview

Machine Learning Operations (MLOps) is a systematic discipline that integrates machine learning, data science, and software engineering practices to automate and streamline the end-to-end ML lifecycle. This lifecycle encompasses data preparation, model training, evaluation, deployment, monitoring, and ongoing maintenance. The goal of MLOps is to ensure that ML models are developed, deployed, and operated reliably, efficiently, and at scale.

To ground the discussion, consider a conventional ML application involving centralized infrastructure. A ridesharing company may aim to predict real-time rider demand using a machine learning model. The data science team might invest significant time designing and training the model. However, when it comes time to deploy it, the model often needs to be reengineered to align with the engineering team's production requirements. This disconnect can introduce weeks of delay and engineering overhead. MLOps addresses this gap.

By establishing standard protocols, tools, and workflows, MLOps enables models developed during experimentation to transition seamlessly into production. It promotes collaboration across traditionally siloed roles, including data scientists, ML engineers, and DevOps professionals, by defining interfaces and responsibilities. MLOps also supports continuous integration and delivery for ML, allowing teams to retrain, validate, and redeploy models frequently in response to new data or system conditions.

Returning to the ridesharing example, a mature MLOps practice would allow the company to continuously retrain its demand forecasting model as new ridership data becomes available. It would also make it easier to evaluate alternative model architectures, deploy experimental updates, and monitor system performance in production—all without disrupting live operations. This agility is critical for maintaining model relevance in dynamic environments.

Beyond operational efficiency, MLOps brings important benefits for governance and accountability. It standardizes the tracking of model versions, data lineage, and configuration parameters, creating a reproducible and auditable trail of ML artifacts. This is essential in highly regulated industries such as healthcare and finance, where model explainability and provenance are fundamental requirements.

Organizations across sectors are adopting MLOps to increase team productivity, reduce time-to-market, and improve the reliability of ML systems. The adoption of MLOps not only enhances model performance and robustness but also enables a sustainable approach to managing ML systems at scale.

This chapter introduces the core motivations and foundational components of MLOps, traces its historical development from DevOps, and outlines the key challenges and practices that guide its adoption in modern ML system design.

13.2 Historical Context

MLOps has its roots in DevOps, a set of practices that combines software development (Dev) and IT operations (Ops) to shorten the development lifecycle and enable the continuous delivery of high-quality software. Both DevOps and MLOps emphasize automation, collaboration, and iterative improvement. However, while DevOps emerged to address challenges in software deployment and operational management, MLOps evolved in response to the unique complexities of machine learning workflows—especially those involving data-driven components (Breck et al. 2020). Understanding this evolution is essential for appreciating the motivations and structure of modern ML systems.

13.2.1 DevOps

The term DevOps was coined in 2009 by Patrick Debois, a consultant and Agile practitioner who organized the first DevOpsDays conference in Ghent, Belgium. DevOps extended the principles of the Agile movement, that emphasized close collaboration among development teams and rapid, iterative releases, by bringing IT operations into the fold.

In traditional software pipelines, development and operations teams often worked in silos, leading to inefficiencies, delays, and misaligned priorities. DevOps emerged as a response, advocating for shared ownership, infrastructure as code, and the use of automation to streamline deployment pipelines. Tools such as Jenkins, Docker, and Kubernetes became foundational to implementing continuous integration and continuous delivery (CI/CD)[0] practices.

DevOps promotes collaboration through automation and feedback loops, aiming to reduce time-to-release and improve software reliability. It established the cultural and technical groundwork for extending similar principles to the ML domain.

[0] Continuous Integration/Continuous Delivery (CI/CD): Practices that automate the software delivery process to ensure a seamless and frequent release cycle.

13.2.2 MLOps

MLOps builds on the DevOps foundation but adapts it to the specific demands of ML system development and deployment. While DevOps focuses on integrating and delivering deterministic software, MLOps must manage non-deterministic, data-dependent workflows. These workflows span data acquisition, preprocessing, model training, evaluation, deployment, and continuous monitoring (see Figure 13.2).

> ℹ Definition of MLOps
>
> **Machine Learning Operations (MLOps)** refers to the *engineering discipline* that manages the *end-to-end lifecycle* of machine learning systems, from *data and model development* to *deployment*, *monitoring*, and _maintenance* in production. MLOps addresses *ML-specific challenges*, such as *data and model versioning*, *continuous retraining*, and *behavior under uncertainty*. It emphasizes *collaborative workflows*, *infrastructure automation*, and *governance* to ensure that systems remain *reliable*, *scalable*, and *auditable* throughout their operational lifespan.

Several recurring challenges in operationalizing machine learning motivated the emergence of MLOps as a distinct discipline. One major concern is data drift, where shifts in input data distributions over time degrade model accuracy. This necessitates continuous monitoring and automated retraining procedures. Equally critical is reproducibility—ML workflows often lack standardized mechanisms to track code, datasets, configurations, and environments, making it difficult to reproduce past experiments (Schelter et al. 2018). The lack of explainability in complex models has further driven demand for tools that increase model transparency and interpretability, particularly in regulated domains.

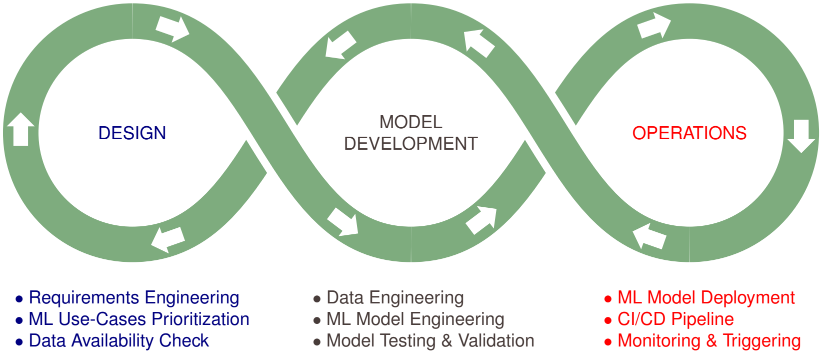

DESIGN
MODEL DEVELOPMENT
OPERATIONS
• Requirements Engineering
• ML Use-Cases Prioritization
• Data Availability Check
• Data Engineering
• ML Model Engineering
• Model Testing & Validation
• ML Model Deployment
• CI/CD Pipeline
• Monitoring & Triggering

Figure 13.2: How MLOps fits in the over all model design and development life cycle.

Post-deployment, many organizations struggle with monitoring model performance in production, especially in detecting silent failures or changes in user behavior. Additionally, the manual overhead involved in retraining and redeploying models creates friction in experimentation and iteration. Finally, configuring and maintaining ML infrastructure is complex and error-prone, highlighting the need for platforms that offer optimized, modular, and reusable infrastructure. Together, these challenges form the foundation for MLOps practices that focus on automation, collaboration, and lifecycle management.

These challenges introduced the need for a new set of tools and workflows tailored to the ML lifecycle. While DevOps primarily unifies software development and IT operations, MLOps requires coordination across a broader set of stakeholders—data scientists, ML engineers, data engineers, and operations teams.

MLOps introduces specialized practices such as data versioning, model versioning, and model monitoring that go beyond the scope of DevOps. It emphasizes scalable experimentation, reproducibility, governance, and responsiveness to evolving data conditions. Table 13.1 summarizes key similarities and differences between DevOps and MLOps:

Table 13.1: Comparison of DevOps and MLOps.

Aspect	DevOps	MLOps
Objective	Streamlining software development and operations processes	Optimizing the lifecycle of machine learning models
Methodology	Continuous Integration and Continuous Delivery (CI/CD) for software development	Similar to CI/CD but focuses on machine learning workflows
Primary Tools	Version control (Git), CI/CD tools (Jenkins, Travis CI), Configuration management (Ansible, Puppet)	Data versioning tools, Model training and deployment tools, CI/CD pipelines tailored for ML
Primary Concerns	Code integration, Testing, Release management, Automation, Infrastructure as code	Data management, Model versioning, Experiment tracking, Model deployment, Scalability of ML workflows
Typical Outcomes	Faster and more reliable software releases, Improved collaboration between development and operations teams	Efficient management and deployment of machine learning models, Enhanced collaboration between data scientists and engineers

These distinctions become clearer when examined through practical examples. One such case study, which focuses on speech recognition, demonstrates the lifecycle of ML deployment and monitoring in action.

13.3 MLOps Key Components

The core components of MLOps form an integrated framework that supports the full machine learning lifecycle—from initial development through deployment and long-term maintenance in production. This section synthesizes key ideas such as automation, reproducibility, and monitoring introduced earlier in the book, while also introducing critical new practices, including governance, model evaluation, and cross-team collaboration. Each component plays a distinct role in creating scalable, reliable, and maintainable ML systems. Together, they form a layered architecture, as illustrated in Figure Figure 13.3, that supports everything from low-level infrastructure to high-level application logic. By understanding how these components interact, practitioners can design systems that are not only performant but also transparent, auditable, and adaptable to changing conditions.

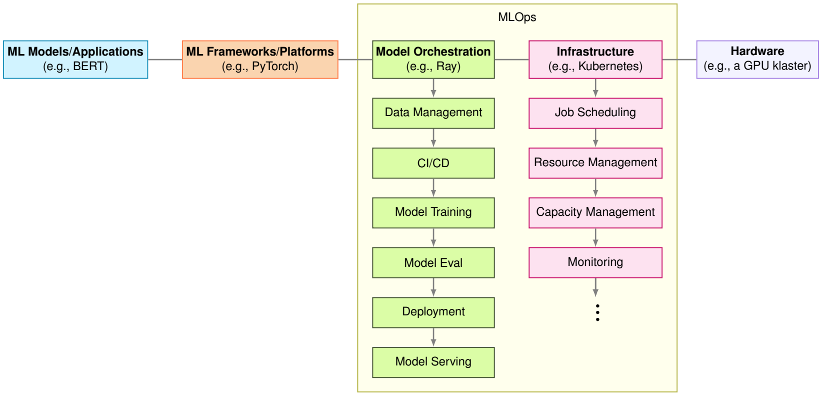


Figure 13.3: The MLOps stack, including ML Models, Frameworks, Model Orchestration, Infrastructure, and Hardware, illustrates the end-to-end workflow of MLOps.

13.3.1 Data Infrastructure and Preparation

Reliable machine learning systems depend on structured, scalable, and repeatable handling of data. From the moment data is ingested to the point where it informs predictions, each stage must preserve quality, consistency, and traceability. In operational settings, data infrastructure supports not only initial development but also continual retraining, auditing, and serving—requiring systems that formalize the transformation and versioning of data throughout the ML lifecycle.

13.3.1.1 Data Management

In earlier chapters, we examined how data is collected, preprocessed, and transformed into features suitable for model training and inference. Within the context of MLOps, these tasks are formalized and scaled into systematic, repeatable processes that ensure data reliability, traceability, and operational efficiency. Data management, in this setting, extends beyond initial preparation to encompass the continuous handling of data artifacts throughout the lifecycle of a machine learning system.

A foundational aspect of MLOps data management is dataset versioning. Machine learning systems often evolve in tandem with the data on which they are trained. Therefore, it is essential to maintain a clear mapping between specific versions of data and corresponding model iterations. Tools such as DVC enable teams to version large datasets alongside code repositories managed by Git, ensuring that data lineage is preserved and that experiments are reproducible.

Supervised learning pipelines also require consistent and well-managed annotation workflows. Labeling tools such as Label Studio support scalable, team-based annotation with integrated audit trails and version histories. These capabilities are particularly important in production settings, where labeling conventions may evolve over time or require refinement across multiple iterations of a project.

In operational environments, data must also be stored in a manner that supports secure, scalable, and collaborative access. Cloud-based object storage systems such as Amazon S3 and Google Cloud Storage offer durability and fine-grained access control, making them well-suited for managing both raw and processed data artifacts. These systems frequently serve as the foundation for downstream analytics, model development, and deployment workflows.

To transition from raw data to analysis- or inference-ready formats, MLOps teams construct automated data pipelines. These pipelines perform structured tasks such as data ingestion, schema validation, deduplication, transformation, and loading. Orchestration tools including Apache Airflow, Prefect, and dbt are commonly used to define and manage these workflows. When managed as code, pipelines support versioning, modularity, and integration with CI/CD systems.

An increasingly important element of the MLOps data infrastructure is the feature store[1]. Feature stores, such as Feast and Tecton, provide a centralized repository for storing and retrieving engineered features. These systems serve both batch and online use cases, ensuring that models access the same feature definitions during training and inference, thereby improving consistency and reducing data leakage.

Consider a predictive maintenance application in an industrial setting. A continuous stream of sensor data is ingested and joined with historical maintenance logs through a scheduled pipeline managed in Airflow. The resulting features, including rolling averages and statistical aggregates, are stored in a feature store for both retraining and low-latency inference. This pipeline is versioned, monitored, and integrated with the model registry, enabling full traceability from data to deployed model predictions.

Effective data management in MLOps is not limited to ensuring data quality. It also establishes the operational backbone that enables model reproducibility, auditability, and sustained deployment at scale. Without robust data management, the integrity of downstream training, evaluation, and serving processes cannot be maintained.

! Important 5: Data Pipelines

Watch on YouTube

Watch on YouTube
Data Pipelines

Scan with your phone
to watch the video

13.3.1.2 Feature Stores

Feature stores provide an abstraction layer between data engineering and machine learning. Their primary purpose is to enable consistent, reliable access to engineered features across training and inference workflows. In conventional pipelines, feature engineering logic may be duplicated, manually reimplemented, or diverge across environments. This introduces risks of training-serving skew[2], data leakage[3], and model drift[4].

Feature stores address these challenges by managing both offline (batch) and online (real-time) feature access in a centralized repository. During training, features are computed and stored in a batch environment—typically in conjunction with historical labels. At inference time, the same transformation logic is applied to fresh data in an online serving system. This architecture ensures that models consume identical features in both contexts, promoting consistency and improving reliability.

[1] Feature Store: A centralized repository for storing, managing, and retrieving feature data used in machine learning models.

[2] Training-serving skew: A discrepancy between model performance during training and inference, often due to differences in data handling.

[3] Data leakage: Occurs when information from outside the training dataset is used to create the model, leading to misleadingly high performance.

[4] Model drift: The change in model performance over time, caused by evolving underlying data patterns.

In addition to enforcing standardization, feature stores support versioning, metadata management, and feature reuse across teams. For example, a fraud detection model and a credit scoring model may rely on overlapping transaction features, which can be centrally maintained, validated, and shared. This reduces engineering overhead and fosters alignment across use cases.

Feature stores can be tightly integrated with data pipelines and model registries, enabling lineage tracking and traceability. When a feature is updated or deprecated, dependent models can be identified and retrained accordingly. This level of integration enhances the operational maturity of ML systems and supports auditing, debugging, and compliance workflows.

13.3.1.3 Versioning and Lineage

Versioning is fundamental to reproducibility and traceability in machine learning systems. Unlike traditional software, ML models depend on multiple changing artifacts—data, feature transformations, model weights, and configuration parameters. To manage this complexity, MLOps practices enforce rigorous tracking of versions across all pipeline components.

Data versioning allows teams to snapshot datasets at specific points in time and associate them with particular model runs. This includes both raw data (e.g., input tables or log streams) and processed artifacts (e.g., cleaned datasets or feature sets). By maintaining a direct mapping between model checkpoints and the data used for training, teams can audit decisions, reproduce results, and investigate regressions.

Model versioning involves registering trained models as immutable artifacts, often alongside metadata such as training parameters, evaluation metrics, and environment specifications. These records are typically maintained in a model registry, which provides a structured interface for promoting, deploying, and rolling back model versions. Some registries also support lineage visualization, which traces the full dependency graph from raw data to deployed prediction.

Together, data and model versioning form the lineage layer of an ML system. This layer enables introspection, experimentation, and governance. When a deployed model underperforms, lineage tools help teams answer questions such as:

- Was the input distribution consistent with training data?
- Did the feature definitions change?
- Is the model version aligned with the serving infrastructure?

By making versioning and lineage first-class citizens in the system design, MLOps enables teams to build and maintain reliable, auditable, and evolvable ML workflows at scale.

13.3.2 Continuous Pipelines and Automation

Automation enables machine learning systems to evolve continuously in response to new data, shifting objectives, and operational constraints. Rather than treating development and deployment as isolated phases, automated pipelines allow for synchronized workflows that integrate data preprocessing, training, evaluation, and release. These pipelines underpin scalable experimentation and ensure the repeatability and reliability of model updates in production.

13.3.2.1 CI/CD Pipelines

In conventional software systems, continuous integration and continuous delivery (CI/CD) pipelines are essential for ensuring that code changes can be tested, validated, and deployed efficiently. In the context of machine learning systems, CI/CD pipelines are adapted to handle additional complexities introduced by data dependencies, model training workflows, and artifact versioning[5]. These pipelines provide a structured mechanism to transition ML models from development into production in a reproducible, scalable, and automated manner.

A typical ML CI/CD pipeline consists of several coordinated stages, including: checking out updated code, preprocessing input data, training a candidate model, validating its performance, packaging the model, and deploying it to a serving environment. In some cases, pipelines also include triggers for automatic retraining based on data drift or performance degradation. By codifying these steps, CI/CD pipelines reduce manual intervention, enforce quality checks, and support continuous improvement of deployed systems.

A wide range of tools is available for implementing ML-focused CI/CD workflows. General-purpose CI/CD orchestrators such as Jenkins, CircleCI, and GitHub Actions are commonly used to manage version control events and execution logic. These tools are frequently integrated with domain-specific platforms such as Kubeflow, Metaflow, and Prefect, which offer higher-level abstractions for managing ML tasks and workflows.

Figure 13.4 illustrates a representative CI/CD pipeline for machine learning systems. The process begins with a dataset and feature repository, from which data is ingested and validated. Validated data is then transformed for model training. A retraining trigger, such as a scheduled job or performance threshold, may initiate this process automatically. Once training and hyperparameter tuning are complete, the resulting model undergoes evaluation against predefined criteria. If the model satisfies the required thresholds, it is registered in a model repository along with metadata, performance metrics, and lineage information. Finally, the model is deployed back into the production system, closing the loop and enabling continuous delivery of updated models.

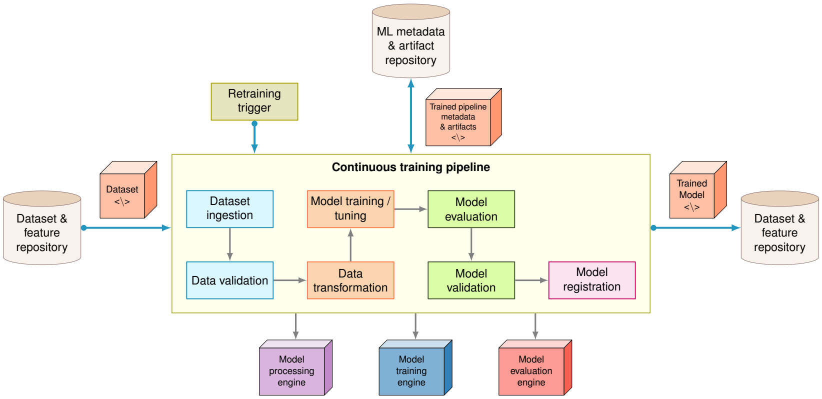


Figure 13.4: MLOps CI/CD diagram. Source: HarvardX.

[5] Artifact Versioning: Managing versions of software artifacts to track changes over time, essential for rollback and understanding dependencies.

As a practical example, consider an image classification model under active development. When a data scientist commits changes to a GitHub repository, a Jenkins pipeline is triggered. The pipeline fetches the latest data, performs preprocessing, and initiates model training. Experiments are tracked using MLflow, which logs metrics and stores model artifacts. After passing automated evaluation tests, the model is containerized and deployed to a staging environment using Kubernetes. If the model meets validation criteria in staging, the pipeline orchestrates a canary deployment, gradually routing production traffic to the new model while monitoring key metrics for anomalies. In case of performance regressions, the system can automatically revert to a previous model version.

CI/CD pipelines play a central role in enabling scalable, repeatable, and safe deployment of machine learning models. By unifying the disparate stages of the ML workflow under continuous automation, these pipelines support faster iteration, improved reproducibility, and greater resilience in production systems. In mature MLOps environments, CI/CD is not an optional layer, but a foundational capability that transforms ad hoc experimentation into a structured and operationally sound development process.

13.3.2.2 Training Pipelines

Model training is a central phase in the machine learning lifecycle, where algorithms are optimized to learn patterns from data. In prior chapters, we introduced the fundamentals of model development and training workflows, including architecture selection, hyperparameter tuning, and evaluation. Within an MLOps context, these activities are reframed as part of a reproducible, scalable, and automated pipeline that supports continual experimentation and reliable production deployment.

Modern machine learning frameworks such as TensorFlow, PyTorch, and Keras provide modular components for building and training models. These libraries include high-level abstractions for layers, activation functions, loss metrics, and optimizers, enabling practitioners to prototype and iterate efficiently. When embedded into MLOps pipelines, these frameworks serve as the foundation for training processes that can be systematically scaled, tracked, and retrained.

Reproducibility is a key objective of MLOps. Training scripts and configurations are version-controlled using tools like Git and hosted on platforms such as GitHub. Interactive development environments, including Jupyter notebooks, are commonly used to encapsulate data ingestion, feature engineering, training routines, and evaluation logic in a unified format. These notebooks can be integrated into automated pipelines, allowing the same logic used for local experimentation to be reused for scheduled retraining in production systems.

Automation further enhances model training by reducing manual effort and standardizing critical steps. MLOps workflows often incorporate techniques such as hyperparameter tuning, neural architecture search, and automatic feature selection to explore the design space efficiently. These tasks are orchestrated using CI/CD pipelines, which automate data preprocessing, model training, evaluation, registration, and deployment. For instance, a Jenkins pipeline may trigger a retraining job when new labeled data becomes available. The resulting model is evaluated against baseline metrics, and if performance thresholds are met, it is deployed automatically.

The increasing availability of cloud-based infrastructure has further expanded the reach of model training. Cloud providers offer managed services[6] that provision high-performance computing resources, which include GPU and TPU accelerators, on demand. Depending on the platform, teams may construct their own training workflows or rely on fully managed services such as Vertex AI Fine Tuning, which support automated adaptation of foundation models to new tasks. Nonetheless, hardware availability, regional access restrictions, and cost constraints remain important considerations when designing cloud-based training systems.

As an illustrative example, consider a data scientist developing a convolutional neural network (CNN) for image classification using a PyTorch notebook. The fastai library is used to simplify model construction and training. The notebook trains the model on a labeled dataset, computes performance metrics, and tunes hyperparameters such as learning rate and architecture depth. Once validated, the training script is version-controlled and incorporated into a retraining pipeline that is periodically triggered based on data updates or model performance monitoring.

Through standardized workflows, versioned environments, and automated orchestration, MLOps enables the model training process to transition from ad hoc experimentation to a robust, repeatable, and scalable system. This not only accelerates development but also ensures that trained models meet production standards for reliability, traceability, and performance.

13.3.2.3 Model Validation

Before a machine learning model is deployed into production, it must undergo rigorous evaluation to ensure that it meets predefined performance, robustness, and reliability criteria. While earlier chapters discussed evaluation in the context of model development, MLOps reframes evaluation as a structured and repeatable process for validating operational readiness. It incorporates practices that support pre-deployment assessment, post-deployment monitoring, and automated regression testing.

The evaluation process typically begins with performance testing against a holdout test set—a dataset not used during training or validation. This dataset is sampled from the same distribution as production data and is used to measure generalization. Core metrics such as accuracy, area under the curve (AUC), precision, recall, and F1 score are computed to quantify model performance. These metrics are not only used at a single point in time but also tracked longitudinally to detect degradation, such as that caused by data drift, where shifts in input distributions can reduce model accuracy over time (see Figure 13.5).

Beyond static evaluation, MLOps encourages controlled deployment strategies that simulate production conditions while minimizing risk. One widely adopted method is canary testing, in which the new model is deployed to a small fraction of users or queries. During this limited rollout, live performance

[6] In cloud computing, managed services involve third-party providers handling infrastructure, application functionalities, and operations.

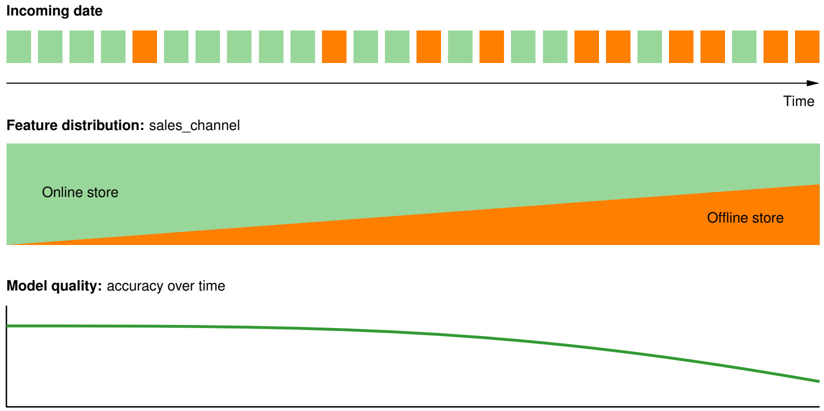


Figure 13.5: Example of data drift over time and how it can impact model performance.

metrics are monitored to assess system stability and user impact. For instance, an e-commerce platform may deploy a new recommendation model to 5% of web traffic and observe metrics such as click-through rate, latency, and prediction accuracy. Only after the model demonstrates consistent and reliable performance is it promoted to full production.

Cloud-based ML platforms further support model evaluation by enabling experiment logging, request replay, and synthetic test case generation. These capabilities allow teams to evaluate different models under identical conditions, facilitating comparisons and root-cause analysis. Tools such as Weights and Biases automate aspects of this process by capturing training artifacts, recording hyperparameter configurations, and visualizing performance metrics across experiments. These tools integrate directly into training and deployment pipelines, improving transparency and traceability.

While automation is central to MLOps evaluation practices, human oversight remains essential. Automated tests may fail to capture nuanced performance issues, such as poor generalization on rare subpopulations or shifts in user behavior. Therefore, teams often combine quantitative evaluation with qualitative review, particularly for models deployed in high-stakes or regulated environments.

In summary, model evaluation within MLOps is a multi-stage process that bridges offline testing and live system monitoring. It ensures that models not only meet technical benchmarks but also behave predictably and responsibly under real-world conditions. These evaluation practices reduce deployment risk and help maintain the reliability of machine learning systems over time.

13.3.3 Model Deployment and Serving

Once a model has been trained and validated, it must be integrated into a production environment where it can deliver predictions at scale. This process involves packaging the model with its dependencies, managing versions, and deploying it in a way that aligns with performance, reliability, and governance requirements. Deployment transforms a sta tic artifact into a live system component. Serving ensures that the model is accessible, reliable, and efficient in responding to inference requests. Together, these components form the bridge between model development and real-world impact.

13.3.3.1 Model Deployment

Teams need to properly package, test, and track ML models to reliably deploy them to production. MLOps introduces frameworks and procedures for actively versioning, deploying, monitoring, and updating models in sustainable ways.

One common approach to deployment involves containerizing models using tools like Docker, which package code, libraries, and dependencies into standardized units. Containers ensure smooth portability across environments, making deployment consistent and predictable. Frameworks like TensorFlow Serving and BentoML help serve predictions from deployed models via performance-optimized APIs. These frameworks handle versioning, scaling, and monitoring.

Before full-scale rollout, teams deploy updated models to staging or QA environments to rigorously test performance. Techniques such as shadow or canary deployments are used to validate new models incrementally. For instance, canary deployments route a small percentage of traffic to the new model while closely monitoring performance. If no issues arise, traffic to the new model gradually increases. Robust rollback procedures are essential to handle unexpected issues, reverting systems to the previous stable model version to ensure minimal disruption. Integration with CI/CD pipelines further automates the deployment and rollback process, enabling efficient iteration cycles.

To maintain lineage and auditability, teams track model artifacts, including scripts, weights, logs, and metrics, using tools like MLflow. Model registries, such as Vertex AI's model registry, act as centralized repositories for storing and managing trained models. These registries not only facilitate version comparisons but also often include access to base models, which may be open source, proprietary, or a hybrid (e.g., LLAMA). Deploying a model from the registry to an inference endpoint is streamlined, handling resource provisioning, model weight downloads, and hosting.

Inference endpoints typically expose the deployed model via REST APIs[7] for real-time predictions. Depending on performance requirements, teams can configure resources, such as GPU accelerators, to meet latency and throughput targets. Some providers also offer flexible options like serverless or batch inference, eliminating the need for persistent endpoints and enabling cost-efficient, scalable deployments. For example, AWS SageMaker Inference supports such configurations. By leveraging these tools and practices, teams can deploy ML models resiliently, ensuring smooth transitions between versions, maintaining production stability, and optimizing performance across diverse use cases.

13.3.3.2 Inference Serving

Once a model has been deployed, the final stage in operationalizing machine learning is to make it accessible to downstream applications or end-users. Serving infrastructure provides the interface between trained models and real-world systems, enabling predictions to be delivered reliably and efficiently. In large-scale settings, such as social media platforms or e-commerce services, serving

[7] REST APIs: Interfaces that allow communication between computer systems over the internet using REST architectural principles.

systems may process tens of trillions of inference queries per day (C.-J. Wu et al. 2019). Meeting such demand requires careful design to balance latency, scalability, and robustness.

To address these challenges, production-grade serving frameworks have emerged. Tools such as TensorFlow Serving, NVIDIA Triton Inference Server, and KServe provide standardized mechanisms for deploying, versioning, and scaling machine learning models across heterogeneous infrastructure. These frameworks abstract many of the lower-level concerns, allowing teams to focus on system behavior, integration, and performance targets.

Model serving architectures are typically designed around three broad paradigms:

1. Online Serving, which provides low-latency, real-time predictions for interactive systems such as recommendation engines or fraud detection.
2. Offline Serving, which processes large batches of data asynchronously, typically in scheduled jobs used for reporting or model retraining.
3. Near-Online (Semi-Synchronous) Serving, which offers a balance between latency and throughput, appropriate for scenarios like chatbots or semi-interactive analytics.

Each of these approaches introduces different constraints in terms of availability, responsiveness, and throughput. Serving systems are therefore constructed to meet specific Service Level Agreements (SLAs) and Service Level Objectives (SLOs), which quantify acceptable performance boundaries along dimensions such as latency, error rates, and uptime. Achieving these goals requires a range of optimizations in request handling, scheduling, and resource allocation.

A number of serving system design strategies are commonly employed to meet these requirements. Request scheduling and batching aggregate inference requests to improve throughput and hardware utilization. For instance, Clipper (Crankshaw et al. 2017) applies batching and caching to reduce response times in online settings. Model instance selection and routing dynamically assign requests to model variants based on system load or user-defined constraints; INFaaS (Romero et al. 2021) illustrates this approach by optimizing accuracy-latency trade-offs across variant models.

1. **Request scheduling and batching**: Efficiently manages incoming ML inference requests, optimizing performance through smart queuing and grouping strategies. Systems like Clipper (Crankshaw et al. 2017) introduce low-latency online prediction serving with caching and batching techniques.
2. **Model instance selection and routing**: Intelligent algorithms direct requests to appropriate model versions or instances. INFaaS (Romero et al. 2021) explores this by generating model-variants and efficiently navigating the trade-off space based on performance and accuracy requirements.
3. **Load balancing**: Distributes workloads evenly across multiple serving instances. MArk (Model Ark) (C. Zhang et al. 2019) demonstrates effective load balancing techniques for ML serving systems.
4. **Model instance autoscaling**: Dynamically adjusts capacity based on demand. Both INFaaS (Romero et al. 2021) and MArk (C. Zhang et al.

2019) incorporate autoscaling capabilities to handle workload fluctuations efficiently.

5. **Model orchestration**: Manages model execution, enabling parallel processing and strategic resource allocation. AlpaServe (Z. Li et al. 2023) demonstrates advanced techniques for handling large models and complex serving scenarios.
6. **Execution time prediction**: Systems like Clockwork (Gujarati et al. 2020) focus on high-performance serving by predicting execution times of individual inferences and efficiently using hardware accelerators.

In more complex inference scenarios, model orchestration coordinates the execution of multi-stage models or distributed components. AlpaServe (Z. Li et al. 2023) exemplifies this by enabling efficient serving of large foundation models through coordinated resource allocation. Finally, execution time prediction enables systems to anticipate latency for individual requests. Clockwork (Gujarati et al. 2020) uses this capability to reduce tail latency and improve scheduling efficiency under high load.

While these systems differ in implementation, they collectively illustrate the critical techniques that underpin scalable and responsive ML-as-a-Service infrastructure. Table 13.2 summarizes these strategies and highlights representative systems that implement them.

Table 13.2: Serving system techniques and example implementations.

Technique	Description	Example System
Request Scheduling & Batching	Groups inference requests to improve throughput and reduce overhead	Clipper
Instance Selection & Routing	Dynamically assigns requests to model variants based on constraints	INFaaS
Load Balancing	Distributes traffic across replicas to prevent bottlenecks	MArk
Autoscaling	Adjusts model instances to match workload demands	INFaaS, MArk
Model Orchestration	Coordinates execution across model components or pipelines	AlpaServe
Execution Time Prediction	Forecasts latency to optimize request scheduling	Clockwork

Together, these strategies form the foundation of robust model serving systems. When effectively integrated, they enable machine learning applications to meet performance targets while maintaining system-level efficiency and scalability.

13.3.4 Infrastructure and Observability

The operational stability of a machine learning system depends on the robustness of its underlying infrastructure. Compute, storage, and networking resources must be provisioned, configured, and scaled to accommodate training workloads, deployment pipelines, and real-time inference. Beyond infrastructure provisioning, effective observability practices ensure that system behavior can be monitored, interpreted, and acted upon as conditions change.

13.3.4.1 Infrastructure Management

Scalable, resilient infrastructure is a foundational requirement for operationalizing machine learning systems. As models move from experimentation to production, MLOps teams must ensure that the underlying computational resources can support continuous integration, large-scale training, automated deployment, and real-time inference. This requires managing infrastructure not as static hardware, but as a dynamic, programmable, and versioned system.

To achieve this, teams adopt the practice of Infrastructure as Code (IaC), which allows infrastructure to be defined, deployed, and maintained using declarative configuration files. Tools such as Terraform, AWS CloudFormation, and Ansible support this paradigm by enabling teams to version infrastructure definitions alongside application code. In MLOps settings, Terraform is widely used to provision and manage resources across public cloud platforms such as AWS, Google Cloud Platform, and Microsoft Azure.

Infrastructure management spans the full lifecycle of ML systems. During model training, teams use IaC scripts to allocate compute instances with GPU or TPU accelerators, configure distributed storage, and deploy container clusters. These configurations ensure that data scientists and ML engineers can access reproducible environments with the required computational capacity. Because infrastructure definitions are stored as code, they can be audited, reused, and integrated into CI/CD pipelines to ensure consistency across environments.

Containerization plays a critical role in making ML workloads portable and consistent. Tools like Docker encapsulate models and their dependencies into isolated units, while orchestration systems such as Kubernetes manage containerized workloads across clusters. These systems enable rapid deployment, resource allocation, and scaling—capabilities that are essential in production environments where workloads can vary dynamically.

To handle changes in workload intensity, including spikes during hyperparameter tuning and surges in prediction traffic, teams rely on cloud elasticity and autoscaling. Cloud platforms support on-demand provisioning and horizontal scaling of infrastructure resources. Autoscaling mechanisms automatically adjust compute capacity based on usage metrics, enabling teams to optimize for both performance and cost-efficiency.

Importantly, infrastructure in MLOps is not limited to the cloud. Many deployments span on-premises, cloud, and edge environments, depending on latency, privacy, or regulatory constraints. A robust infrastructure management strategy must accommodate this diversity by offering flexible deployment targets and consistent configuration management across environments.

To illustrate, consider a scenario in which a team uses Terraform to deploy a Kubernetes cluster on Google Cloud Platform. The cluster is configured to host containerized TensorFlow models that serve predictions via HTTP APIs. As user demand increases, Kubernetes automatically scales the number of pods to handle the load. Meanwhile, CI/CD pipelines update the model containers based on retraining cycles, and monitoring tools track cluster performance, latency, and resource utilization. All infrastructure components, ranging from network configurations to compute quotas, are managed as version-controlled code, ensuring reproducibility and auditability.

By adopting Infrastructure as Code, leveraging cloud-native orchestration, and supporting automated scaling, MLOps teams gain the ability to provision and maintain the resources required for machine learning at production scale. This infrastructure layer underpins the entire MLOps stack, enabling reliable training, deployment, and serving workflows.

13.3.4.2 Monitoring Systems

Monitoring is a critical function in MLOps, enabling teams to maintain operational visibility over machine learning systems deployed in production. Once a model is live, it becomes exposed to real-world inputs, evolving data distributions, and shifting user behavior. Without continuous monitoring, it becomes difficult to detect performance degradation, data quality issues, or system failures in a timely manner.

Effective monitoring spans both model behavior and infrastructure performance. On the model side, teams track metrics such as accuracy, precision, recall, and the confusion matrix using live or sampled predictions. By evaluating these metrics over time, they can detect whether the model's performance remains stable or begins to drift.

One of the primary risks in production ML systems is model drift—a gradual decline in model performance as the input data distribution or the relationship between inputs and outputs changes. Drift manifests in two main forms:

- Concept drift occurs when the underlying relationship between features and targets evolves. For example, during the COVID-19 pandemic, purchasing behavior shifted dramatically, invalidating many previously accurate recommendation models.
- Data drift refers to shifts in the input data distribution itself. In applications such as self-driving cars, this may result from seasonal changes in weather, lighting, or road conditions, all of which affect the model's inputs.

In addition to model-level monitoring, infrastructure-level monitoring tracks indicators such as CPU and GPU utilization, memory and disk consumption, network latency, and service availability. These signals help ensure that the system remains performant and responsive under varying load conditions. Tools such as Prometheus, Grafana, and Elastic are widely used to collect, aggregate, and visualize operational metrics. These tools often integrate into dashboards that offer real-time and historical views of system behavior.

Proactive alerting mechanisms are configured to notify teams when anomalies or threshold violations occur. For example, a sustained drop in model accuracy may trigger an alert to investigate potential drift, prompting retraining with updated data. Similarly, infrastructure alerts can signal memory saturation or degraded network performance, allowing engineers to take corrective action before failures propagate.

Ultimately, robust monitoring enables teams to detect problems before they escalate, maintain high service availability, and preserve the reliability and trustworthiness of machine learning systems. In the absence of such practices, models may silently degrade or systems may fail under load, undermining the effectiveness of the ML pipeline as a whole.

Watch on YouTube
Model Monitoring

Scan with your phone
to watch the video

! Important 6: Model Monitoring

🖥 Watch on YouTube

13.3.5 Governance and Collaboration

13.3.5.1 Model Governance

As machine learning systems become increasingly embedded in decision-making processes, governance has emerged as a critical pillar of MLOps. Governance refers to the policies, practices, and tools used to ensure that models are transparent, fair, accountable, and compliant with ethical standards and regulatory requirements. Without proper governance, deployed models may produce biased or opaque decisions, leading to significant legal, reputational, and societal risks.

Governance begins during the model development phase, where teams implement techniques to increase transparency and explainability. For example, methods such as SHAP and LIME offer post hoc explanations of model predictions by identifying which input features were most influential in a particular decision. These techniques allow auditors, developers, and non-technical stakeholders to better understand how and why a model behaves the way it does.

In addition to interpretability, fairness is a central concern in governance. Bias detection tools analyze model outputs across different demographic groups, including those defined by age, gender, or ethnicity, to identify disparities in performance. For instance, a model used for loan approval should not systematically disadvantage certain populations. MLOps teams employ pre-deployment audits on curated, representative datasets to evaluate fairness, robustness, and overall model behavior before a system is put into production.

Governance also extends into the post-deployment phase. As introduced in the previous section on monitoring, teams must track for concept drift, where the statistical relationships between features and labels evolve over time. Such drift can undermine the fairness or accuracy of a model, particularly if the shift disproportionately affects a specific subgroup. By analyzing logs and user feedback, teams can identify recurring failure modes, unexplained model outputs, or emerging disparities in treatment across user segments.

Supporting this lifecycle approach to governance are platforms and toolkits that integrate governance functions into the broader MLOps stack. For example, Watson OpenScale provides built-in modules for explainability, bias detection, and monitoring. These tools allow governance policies to be encoded as part of automated pipelines, ensuring that checks are consistently applied throughout development, evaluation, and production.

Ultimately, governance focuses on three core objectives: transparency, fairness, and compliance. Transparency ensures that models are interpretable and auditable. Fairness promotes equitable treatment across user groups. Compliance ensures alignment with legal and organizational policies. Embedding governance practices throughout the MLOps lifecycle transforms machine learning from a technical artifact into a trustworthy system capable of serving societal and organizational goals.

13.3.5.2 Cross-Functional Collaboration

Machine learning systems are developed and maintained by multidisciplinary teams, including data scientists, ML engineers, software developers, infrastructure specialists, product managers, and compliance officers. As these roles span different domains of expertise, effective communication and collaboration are essential to ensure alignment, efficiency, and system reliability. MLOps fosters this cross-functional integration by introducing shared tools, processes, and artifacts that promote transparency and coordination across the machine learning lifecycle.

Collaboration begins with consistent tracking of experiments, model versions, and metadata. Tools such as MLflow provide a structured environment for logging experiments, capturing parameters, recording evaluation metrics, and managing trained models through a centralized registry. This registry serves as a shared reference point for all team members, enabling reproducibility and easing handoff between roles. Integration with version control systems such as GitHub and GitLab further streamlines collaboration by linking code changes with model updates and pipeline triggers.

In addition to tracking infrastructure, teams benefit from platforms that support exploratory collaboration. Weights & Biases is one such platform that allows data scientists to visualize experiment metrics, compare training runs, and share insights with peers. Features such as live dashboards and experiment timelines facilitate discussion and decision-making around model improvements, hyperparameter tuning, or dataset refinements. These collaborative environments reduce friction in model development by making results interpretable and reproducible across the team.

Beyond model tracking, collaboration also depends on shared understanding of data semantics and usage. Establishing common data contexts, by means of glossaries, data dictionaries, schema references, and lineage documentation, ensures that all stakeholders interpret features, labels, and statistics consistently. This is particularly important in large organizations, where data pipelines may evolve independently across teams or departments.

For example, a data scientist working on an anomaly detection model may use Weights & Biases to log experiment results and visualize performance trends. These insights are shared with the broader team to inform feature engineering decisions. Once the model reaches an acceptable performance threshold, it is registered in MLflow along with its metadata and training lineage. This allows an ML engineer to pick up the model for deployment without ambiguity about its provenance or configuration.

By integrating collaborative tools, standardized documentation, and transparent experiment tracking, MLOps removes communication barriers that have traditionally slowed down ML workflows. It enables distributed teams to operate cohesively, accelerating iteration cycles and improving the reliability of deployed systems.

Watch on YouTube
Deployment Challenges

Scan with your phone
to watch the video

! Important 7: Deployment Challenges

Watch on YouTube

13.4 Hidden Technical Debt

As machine learning systems mature and scale, they often accumulate technical debt—the long-term cost of expedient design decisions made during development. Originally proposed in software engineering in the 1990s, the technical debt metaphor compares shortcuts in implementation to financial debt: it may enable short-term velocity, but requires ongoing interest payments in the form of maintenance, refactoring, and systemic risk. While some debt is strategic and manageable, uncontrolled technical debt can inhibit flexibility, slow iteration, and introduce brittleness into production systems.

In machine learning, technical debt takes on new and less visible forms, arising not only from software abstractions but also from data dependencies, model entanglement, feedback loops, and evolving operational environments. The complexity of ML systems, which encompass data ingestion, feature extraction, training pipelines, and deployment infrastructure, makes them especially prone to hidden forms of debt (Sculley et al. 2015).

Figure 13.6 provides a conceptual overview of the relative size and interdependence of components in an ML system. The small black box in the center represents the model code itself—a surprisingly small portion of the overall system. Surrounding it are much larger components: configuration, data collection, and feature engineering. These areas, though often overlooked, are critical to system functionality and are major sources of technical debt when poorly designed or inconsistently maintained.

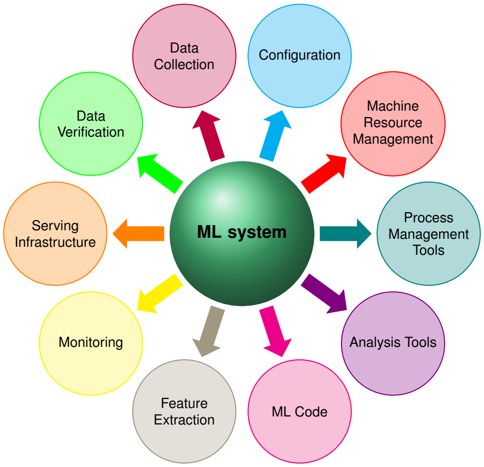


Figure 13.6: ML system components. Source: Sculley et al. (2015)

The sections that follow describe key categories of technical debt unique to ML systems. Each subsection highlights common sources, illustrative examples, and potential mitigations. While some forms of debt may be unavoidable during early development, understanding their causes and impact is essential for building robust and maintainable ML systems.

13.4.1 Boundary Erosion

In traditional software systems, modularity and abstraction provide clear boundaries between components, allowing changes to be isolated and behavior to remain predictable. Machine learning systems, in contrast, tend to blur these boundaries. The interactions between data pipelines, feature engineering, model training, and downstream consumption often lead to tightly coupled components with poorly defined interfaces.

This erosion of boundaries makes ML systems particularly vulnerable to cascading effects from even minor changes. A seemingly small update to a preprocessing step or feature transformation can propagate through the system in unexpected ways, breaking assumptions made elsewhere in the pipeline. This lack of encapsulation increases the risk of entanglement, where dependencies between components become so intertwined that local modifications require global understanding and coordination.

One manifestation of this problem is known as CACE—"Changing Anything Changes Everything." When systems are built without strong boundaries, adjusting a feature encoding, model hyperparameter, or data selection criterion can affect downstream behavior in unpredictable ways. This inhibits iteration and makes testing and validation more complex. For example, changing the binning strategy of a numerical feature may cause a previously tuned model to underperform, triggering retraining and downstream evaluation changes.

To mitigate boundary erosion, teams should prioritize architectural practices that support modularity and encapsulation. Designing components with well-defined interfaces allows teams to isolate faults, reason about changes, and reduce the risk of system-wide regressions. For instance, clearly separating data ingestion from feature engineering, and feature engineering from modeling logic, introduces layers that can be independently validated, monitored, and maintained.

Boundary erosion is often invisible in early development but becomes a significant burden as systems scale or require adaptation. Proactive design decisions that preserve abstraction and limit interdependencies are essential to managing complexity and avoiding long-term maintenance costs.

13.4.2 Correction Cascades

As machine learning systems evolve, they often undergo iterative refinement to address performance issues, accommodate new requirements, or adapt to environmental changes. In well-engineered systems, such updates are localized and managed through modular changes. However, in ML systems, even small adjustments can trigger correction cascades—a sequence of dependent fixes that propagate backward and forward through the workflow.

Figure 13.7 illustrates how these cascades emerge across different stages of the ML lifecycle, from problem definition and data collection to model development and deployment. Each arc represents a corrective action, and the colors indicate different sources of instability, including inadequate domain expertise, brittle real-world interfaces, misaligned incentives, and insufficient documentation. The red arrows represent cascading revisions, while the dotted arrow at the bottom highlights a full system restart—a drastic but sometimes necessary outcome.

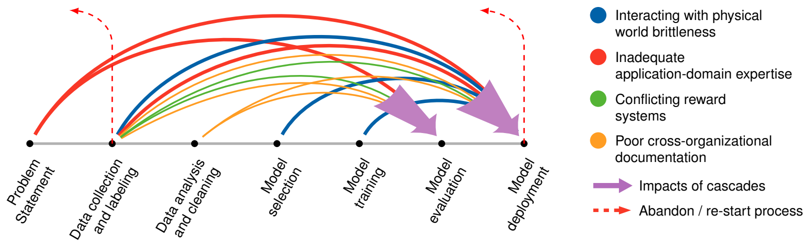

Problem Statement
Data collection and labeling
Data analysis and cleaning
Model selection
Model training
Model evaluation
Model deployment
Interacting with physical world brittleness
Inadequate application-domain expertise
Conflicting reward systems
Poor cross-organizational documentation
Impacts of cascades
Abandon / re-start process

Figure 13.7: Correction cascades flowchart.

One common source of correction cascades is sequential model development—reusing or fine-tuning existing models to accelerate development for new tasks. While this strategy is often efficient, it can introduce hidden dependencies that are difficult to unwind later. Assumptions baked into earlier models become implicit constraints for future models, limiting flexibility and increasing the cost of downstream corrections.

Consider a scenario where a team fine-tunes a customer churn prediction model for a new product. The original model may embed product-specific behaviors or feature encodings that are not valid in the new setting. As performance issues emerge, teams may attempt to patch the model, only to discover that the true problem lies several layers upstream—perhaps in the original feature selection or labeling criteria.

To avoid or reduce the impact of correction cascades, teams must make careful tradeoffs between reuse and redesign. Several factors influence this decision. For small, static datasets, fine-tuning may be appropriate. For large or rapidly evolving datasets, retraining from scratch provides greater control and adaptability. Fine-tuning also requires fewer computational resources, making it attractive in constrained settings. However, modifying foundational components later becomes extremely costly due to these cascading effects.

Therefore, careful consideration should be given to introducing fresh model architectures, even if resource-intensive, to avoid correction cascades down the line. This approach may help mitigate the amplifying effects of issues downstream and reduce technical debt. However, there are still scenarios where sequential model building makes sense, necessitating a thoughtful balance between efficiency, flexibility, and long-term maintainability in the ML development process.

13.4.3 Undeclared Consumers

Machine learning systems often provide predictions or outputs that serve as inputs to other services, pipelines, or downstream models. In traditional software, these connections are typically made explicit through APIs, service contracts, or documented dependencies. In ML systems, however, it is common for model outputs to be consumed by undeclared consumers—downstream components that rely on predictions without being formally tracked or validated.

This lack of visibility introduces a subtle but serious form of technical debt. Because these consumers are not declared or governed by explicit interfaces, updates to the model, including changes in output format, semantics, or feature behavior, can silently break downstream functionality. The original model was not designed with these unknown consumers in mind, so its evolution risks unintended consequences across the broader system.

The situation becomes more problematic when these downstream consumers feed back into the original model's training data. This introduces feedback loops that are difficult to detect and nearly impossible to reason about analytically. For instance, if a model's output is used in a recommendation system and user behavior is influenced by those recommendations, future training data becomes contaminated by earlier predictions. Such loops can distort model behavior, create self-reinforcing biases, and mask performance regressions.

One example might involve a credit scoring model whose outputs are consumed by a downstream eligibility engine. If the eligibility system later influences which applicants are accepted, which, in turn, affects the label distribution in the next training cycle, the model is now shaping the very data on which it will be retrained.

To mitigate the risks associated with undeclared consumers, teams should begin by implementing strict access controls to limit who or what can consume model outputs. Rather than making predictions broadly available, systems should expose outputs only through well-defined interfaces, ensuring that their use can be monitored and audited. In addition, establishing formal interface contracts, which include documented schemas, value ranges, and semantic expectations, helps enforce consistent behavior across components and reduces the likelihood of misinterpretation. Monitoring and logging mechanisms can provide visibility into where and how predictions are used, revealing dependencies that may not have been anticipated during development. Finally, architectural decisions should emphasize system boundaries that encapsulate model behavior, thereby isolating changes and minimizing the risk of downstream entanglement. Together, these practices support a more disciplined and transparent approach to system integration, reducing the likelihood of costly surprises as ML systems evolve.

13.4.4 Data Dependency Debt

Machine learning systems rely heavily on data pipelines that ingest, transform, and deliver training and inference inputs. Over time, these pipelines often develop implicit and unstable dependencies that become difficult to trace, validate, or manage—leading to what is known as data dependency debt. This form of debt is particularly challenging because it tends to accumulate silently and may only become visible when a downstream model fails unexpectedly due to changes in upstream data.

In traditional software systems, compilers, static analysis tools, and dependency checkers help engineers track and manage code-level dependencies.

These tools enable early detection of unused imports, broken interfaces, and type mismatches. However, ML systems typically lack equivalent tooling for analyzing data dependencies, which include everything from feature generation scripts and data joins to external data sources and labeling conventions. Without such tools, changes to even a single feature or schema can ripple across a system without warning.

Two common forms of data dependency debt are unstable inputs and underutilized inputs. Unstable inputs refer to data sources that change over time, whether in content, structure, or availability, leading to inconsistent model behavior. A model trained on one version of a feature may produce unexpected results when that feature's distribution or encoding changes. Underutilized inputs refer to data elements included in training pipelines that have little or no impact on model performance. These features increase complexity, slow down processing, and increase the surface area for bugs, yet provide little return on investment.

One approach to managing unstable dependencies is to implement robust data versioning. By tracking which data snapshot was used for training a given model, teams can reproduce results and isolate regressions. However, versioning also introduces overhead: multiple versions must be stored, managed, and tested for staleness. For underutilized inputs, a common strategy is to run leave-one-feature-out evaluations, where features are systematically removed to assess their contribution to model performance. This analysis can guide decisions about whether to simplify the feature set or deprecate unused data streams.

Addressing data dependency debt requires both architectural discipline and appropriate tooling. ML systems must be designed with traceability in mind—recording not just what data was used, but where it came from, how it was transformed, and how it affected model behavior. For example, consider an e-commerce platform that includes a "days since last login" feature in its churn prediction model. If the meaning of this feature changes, for instance, if a platform redesign results in users being automatically logged in through a companion app, the input distribution will shift, potentially degrading model performance. Without explicit tracking and validation of this data dependency, the issue might go unnoticed until accuracy metrics decline in production. As systems scale, unexamined data dependencies like these become a major source of brittleness and drift. Investing in structured data practices early in the lifecycle, including schema validation, lineage tracking, and dependency testing, can help prevent these issues from compounding over time.

13.4.5 Feedback Loops

Unlike traditional software systems, machine learning models have the capacity to influence their own future behavior through the data they help generate. This dynamic creates feedback loops, where model predictions shape future inputs, often in subtle and difficult-to-detect ways. When unaddressed, these loops introduce a unique form of technical debt: the inability to analyze and reason about model behavior over time, leading to what is known as feedback loop analysis debt.

Feedback loops in ML systems can be either direct or indirect. A direct feedback loop occurs when a model's outputs directly affect future training data. For example, in an online recommendation system, the items a model suggests may strongly influence user clicks and, consequently, the labeled data used for retraining. If the model consistently promotes a narrow subset of items, it may bias the training set over time, reinforcing its own behavior and reducing exposure to alternative signals.

Indirect or hidden feedback loops arise when two or more systems interact with one another, frequently via real-world processes, without clear visibility into their mutual influence. For instance, consider two separate ML models deployed by a financial institution: one predicts credit risk, and the other recommends credit offers. If the output of the second model implicitly affects the population that is later scored by the first, a feedback loop is created without any explicit connection between the two systems. These loops are especially dangerous because they bypass traditional validation frameworks and may take weeks or months to manifest.

Feedback loops undermine assumptions about data independence and stationarity. They can mask model degradation, introduce long-term bias, and lead to unanticipated performance failures. Because most ML validation is performed offline with static datasets, these dynamic interactions are difficult to detect before deployment.

Several mitigation strategies exist, though none are comprehensive. Careful monitoring of model performance across cohorts and over time can help reveal the emergence of loop-induced drift. Canary deployments[8] allow teams to test new models on a small subset of traffic and observe behavior before full rollout. More fundamentally, architectural practices that reduce coupling between system components, including the isolation of decision-making logic from user-facing outcomes, can help minimize the propagation of influence.

Ultimately, feedback loops reflect a deeper challenge in ML system design: models do not operate in isolation, but in dynamic environments where their outputs alter future inputs. Reducing analysis debt[9] requires designing systems with these dynamics in mind and embedding mechanisms to detect and manage self-influencing behavior over time.

13.4.6 Pipeline Debt

As machine learning workflows grow in scope, teams often assemble pipelines that stitch together multiple components—data ingestion, feature extraction, model training, evaluation, and deployment. In the absence of standard interfaces or modular abstractions, these pipelines tend to evolve into ad hoc constructions of custom scripts, manual processes, and undocumented assumptions. Over time, this leads to pipeline debt: a form of technical debt arising from complexity, fragility, and a lack of reusability in ML workflows.

This problem is often described as the emergence of a "pipeline jungle," where modifications become difficult, and experimentation is constrained by brittle interdependencies. When teams are reluctant to refactor fragile pipelines, they resort to building alternate versions for new use cases or experiments. As these variations accumulate, so do inconsistencies in data processing, met-

[8] Canary deployment: A strategy to reduce risk by rolling out changes to a small subset of users before full-scale implementation.

[9] The effort to manage and mitigate increases in complexity and costs in understanding and modifying systems over time.

ric computation, and configuration management. The result is duplication, reduced efficiency, and a growing risk of errors.

Consider a real-world scenario where a team maintains multiple models that rely on different but overlapping preprocessing pipelines. One model applies text normalization using simple lowercasing, while another uses a custom tokenization library. Over time, discrepancies emerge in behavior, leading to conflicting evaluation metrics and unexpected model drift. As new models are introduced, developers are unsure which pipeline to reuse or modify, and duplications multiply.

Pipeline debt also limits collaboration across teams. Without well-defined interfaces or shared abstractions, it becomes difficult to exchange components or adopt best practices. Team members often need to reverse-engineer pipeline logic, slowing onboarding and increasing the risk of introducing regressions.

The most effective way to manage pipeline debt is to embrace modularity and encapsulation. Well-architected pipelines define clear inputs, outputs, and transformation logic, often expressed through workflow orchestration tools such as Apache Airflow, Prefect, or Kubeflow Pipelines. These tools help teams formalize processing steps, track lineage, and monitor execution.

In addition, the adoption of shared libraries for feature engineering, transformation functions, and evaluation metrics promotes consistency and reuse. Teams can isolate logic into composable units that can be independently tested, versioned, and integrated across models. This reduces the risk of technical lock-in and enables more agile development as systems evolve.

Ultimately, pipeline debt reflects a breakdown in software engineering rigor applied to ML workflows. Investing in interfaces, documentation, and shared tooling not only improves maintainability but also unlocks faster experimentation and system scalability.

13.4.7 Configuration Debt

Configuration is a critical yet often undervalued component of machine learning systems. Tuning parameters such as learning rates, regularization strengths, model architectures, feature processing options, and evaluation thresholds all require deliberate management. However, in practice, configurations are frequently introduced in an ad hoc manner—manually adjusted during experimentation, inconsistently documented, and rarely versioned. This leads to the accumulation of configuration debt: the technical burden resulting from fragile, opaque, and outdated settings that undermine system reliability and reproducibility.

When configuration debt accumulates, several challenges emerge. Fragile configurations may contain implicit assumptions about data distributions, training schedules, or pipeline structure that no longer hold as the system evolves. In the absence of proper documentation, these assumptions become embedded in silent defaults—settings that function in development but fail in production. Teams may hesitate to modify these configurations out of fear of introducing regressions, further entrenching the problem. Additionally, when configurations are not centrally tracked, knowledge about what parameters work well becomes siloed within individuals or specific notebooks, leading to redundant experimentation and slowed iteration.

For example, consider a team deploying a neural network for customer segmentation. During development, one data scientist improves performance by tweaking several architectural parameters, by adding layers, changing activation functions, and adjusting batch sizes, but these changes are stored locally and never committed to the shared configuration repository. Months later, the model is retrained on new data, but the performance degrades unexpectedly. Without a consistent record of previous configurations, the team struggles to identify what changed. The lack of traceability not only delays debugging but also undermines confidence in the reproducibility of prior results.

Mitigating configuration debt requires integrating configuration management into the ML system lifecycle. Teams should adopt structured formats, including YAML, JSON, and domain-specific configuration frameworks, and store them in version-controlled repositories alongside model code. Validating configurations as part of the training and deployment process ensures that unexpected or invalid parameter settings are caught early. Automated tools for hyperparameter optimization and neural architecture search further reduce reliance on manual tuning and help standardize configuration discovery.

Above all, ML systems benefit when configuration is treated not as a side effect of experimentation, but as a first-class system component. Like code, configurations must be tested, documented, and maintained. Doing so enables faster iteration, easier debugging, and more reliable system behavior over time.

13.4.8 Early-Stage Debt

In the early phases of machine learning development, teams often move quickly to prototype models, experiment with data sources, and explore modeling approaches. During this stage, speed and flexibility are critical, and some level of technical debt is expected and even necessary to support rapid iteration. However, the decisions made in these early stages, particularly when motivated by urgency instead of design, can introduce early-stage debt that becomes increasingly difficult to manage as the system matures.

This form of debt often stems from shortcuts in code organization, data preprocessing, feature engineering, or model packaging. Pipelines may be built without clear abstractions, evaluation scripts may lack reproducibility, and configuration files may be undocumented or fragmented. While such practices may be justified in the exploratory phase, they become liabilities once the system enters production or needs to scale across teams and use cases.

For example, a startup team developing a minimum viable product (MVP) might embed core business logic directly into the model training code—such as applying customer-specific rules or filters during preprocessing. This expedites initial experimentation but creates a brittle system in which modifying the business logic or model behavior requires untangling deeply intertwined code. As the company grows and multiple teams begin working on the system, these decisions limit flexibility, slow iteration, and increase the risk of breaking core functionality during updates.

Despite these risks, not all early-stage debt is harmful. The key distinction lies in whether the system is designed to support evolution. Techniques such as using modular code, isolating configuration from logic, and containerizing experimental environments allow teams to move quickly without sacrificing future maintainability. Abstractions, including shared data access layers and feature transformation modules, can be introduced incrementally as patterns stabilize.

To manage early-stage debt effectively, teams should adopt the principle of flexible foundations: designing for change without over-engineering. This means identifying which components are likely to evolve and introducing appropriate boundaries and interfaces early on. As the system matures, natural inflection points emerge—opportunities to refactor or re-architect without disrupting existing workflows.

Accepting some technical debt in the short term is often a rational tradeoff. The challenge is ensuring that such debt is intentional, tracked, and revisited before it becomes entrenched. By investing in adaptability from the beginning, ML teams can balance early innovation with long-term sustainability.

13.4.9 Real-World Examples

Hidden technical debt is not just theoretical—it has played a critical role in shaping the trajectory of real-world machine learning systems. These examples illustrate how unseen dependencies and misaligned assumptions can accumulate quietly, only to become major liabilities over time:

13.4.9.1 YouTube's Recommendation System and Feedback Loops

YouTube's recommendation engine has faced repeated criticism for promoting sensational or polarizing content. A large part of this stems from feedback loop debt: recommendations influence user behavior, which in turn becomes training data. Over time, this led to unintended content amplification. Mitigating this required substantial architectural overhauls, including cohort-based evaluation, delayed labeling, and more explicit disentanglement between engagement metrics and ranking logic.

13.4.9.2 Zillow's "Zestimate" and Correction Cascades

Zillow's home valuation model (Zestimate) faced significant correction cascades during its iBuying venture. When initial valuation errors propagated into purchasing decisions, retroactive corrections triggered systemic instability that required data revalidation, model redesign, and eventually a full system rollback. The company shut down the iBuying arm in 2021, citing model unpredictability and data feedback effects as core challenges.

13.4.9.3 Tesla Autopilot and Undeclared Consumers

In early deployments, Tesla's Autopilot made driving decisions based on models whose outputs were repurposed across subsystems without clear boundaries. Over-the-air updates occasionally introduced silent behavior changes that affected multiple subsystems (e.g., lane centering and braking) in unpredictable ways. This entanglement illustrates undeclared consumer debt and the risks of skipping strict interface governance in ML-enabled safety-critical systems.

13.4.9.4 Facebook's News Feed and Configuration Debt

Facebook's News Feed algorithm has undergone numerous iterations, often driven by rapid experimentation. However, the lack of consistent configuration management led to opaque settings that influenced content ranking without clear documentation. As a result, changes to the algorithm's behavior were difficult to trace, and unintended consequences emerged from misaligned configurations. This situation highlights the importance of treating configuration as a first-class citizen in ML systems.

13.4.10 Managing Hidden Technical Debt

While the examples discussed highlight the consequences of hidden technical debt in large-scale systems, they also offer valuable lessons for how such debt can be surfaced, controlled, and ultimately reduced. Managing hidden debt requires more than reactive fixes—it demands a deliberate and forward-looking approach to system design, team workflows, and tooling choices.

A foundational principle is to treat data and configuration as integral parts of the system architecture, not as peripheral artifacts. As shown in Figure 13.6, the bulk of an ML system lies outside the model code itself—in components like feature engineering, configuration, monitoring, and serving infrastructure. These surrounding layers often harbor the most persistent forms of debt, particularly when changes are made without systematic tracking or validation.

Versioning data transformations, labeling conventions, and training configurations enables teams to reproduce past results, localize regressions, and understand the impact of design choices over time. Tools that enable this, such as DVC for data versioning, Hydra for configuration management, and MLflow for experiment tracking, help ensure that the system remains traceable as it evolves. Importantly, version control must extend beyond the model checkpoint to include the data and configuration context in which it was trained and evaluated.

Another key strategy is encapsulation through modular interfaces. The cascading failures seen in tightly coupled systems highlight the importance of defining clear boundaries between components. Without well-specified APIs or contracts, changes in one module can ripple unpredictably through others. By contrast, systems designed around loosely coupled components, in which each module has well-defined responsibilities and limited external assumptions, are far more resilient to change.

Encapsulation also supports dependency awareness, reducing the likelihood of undeclared consumers silently reusing outputs or internal representations. This is especially important in feedback-prone systems, where hidden dependencies can introduce behavioral drift over time. Exposing outputs through audited, documented interfaces makes it easier to reason about their use and to trace downstream effects when models evolve.

Observability and monitoring further strengthen a system's defenses against hidden debt. While static validation may catch errors during development, many forms of ML debt only manifest during deployment, especially in dynamic environments. Monitoring distribution shifts, feature usage patterns, and cohort-specific performance metrics helps detect degradation early, before it impacts users or propagates into future training data. Canary deployments and progressive rollouts are essential tools for limiting risk while allowing systems to evolve.

Teams should also invest in institutional practices that periodically surface and address technical debt. Debt reviews, pipeline audits, and schema validation sprints serve as checkpoints where teams step back from rapid iteration and assess the system's overall health. These reviews create space for refactoring, pruning unused features, consolidating redundant logic, and reasserting boundaries that may have eroded over time.

Finally, the management of technical debt must be aligned with a broader cultural commitment to maintainability. This means prioritizing long-term system integrity over short-term velocity, especially once systems reach maturity or are integrated into critical workflows. It also means recognizing when debt is strategic, which is incurred deliberately to facilitate exploration, and ensuring it is tracked and revisited before it becomes entrenched.

In all cases, managing hidden technical debt is not about eliminating complexity, but about designing systems that can accommodate it without becoming brittle. Through architectural discipline, thoughtful tooling, and a willingness to refactor, ML practitioners can build systems that remain flexible and reliable, even as they scale and evolve.

13.4.11 Summary

Technical debt in machine learning systems is both pervasive and distinct from debt encountered in traditional software engineering. While the original metaphor of financial debt highlights the tradeoff between speed and long-term cost, the analogy falls short in capturing the full complexity of ML systems. In machine learning, debt often arises not only from code shortcuts but also from entangled data dependencies, poorly understood feedback loops, fragile pipelines, and configuration sprawl. Unlike financial debt, which can be explicitly quantified, ML technical debt is largely hidden, emerging only as systems scale, evolve, or fail.

This chapter has outlined several forms of ML-specific technical debt, each rooted in different aspects of the system lifecycle. Boundary erosion undermines modularity and makes systems difficult to reason about. Correction cascades illustrate how local fixes can ripple through a tightly coupled workflow. Undeclared consumers and feedback loops introduce invisible dependencies that challenge traceability and reproducibility. Data and configuration debt reflect the fragility of inputs and parameters that are poorly managed, while pipeline and change adaptation debt expose the risks of inflexible architectures. Early-stage debt reminds us that even in the exploratory phase, decisions should be made with an eye toward future extensibility.

The common thread across all these debt types is the need for system-level thinking. ML systems are not just code—they are evolving ecosystems of data, models, infrastructure, and teams. Managing technical debt requires architectural discipline, robust tooling, and a culture that values maintainability alongside innovation. It also requires humility: acknowledging that today's solutions may become tomorrow's constraints if not designed with care.

As machine learning becomes increasingly central to production systems, understanding and addressing hidden technical debt is essential. Doing so not only improves reliability and scalability, but also empowers teams to iterate faster, collaborate more effectively, and sustain the long-term evolution of their systems.

13.5 Roles and Responsibilities

Operationalizing machine learning systems requires coordinated contributions from professionals with diverse technical and organizational expertise. Unlike traditional software engineering workflows, machine learning introduces additional complexity through its reliance on dynamic data, iterative experimentation, and probabilistic model behavior. As a result, no single role can independently manage the end-to-end machine learning lifecycle. Figure 13.8 provides a high level overview of how these roles relate to each other.

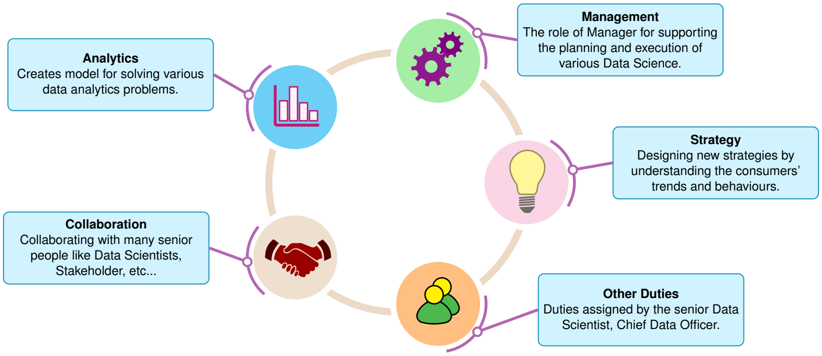

Management
The role of Manager for supporting the planning and execution of various Data Science.
Analytics
Creates model for solving various data analytics problems.
Strategy
Designing new strategies by understanding the consumers' trends and behaviours.
Collaboration
Collaborating with many senior people like Data Scientists, Stakeholder, etc...
Other Duties
Duties assigned by the senior Data Scientist, Chief Data Officer.

Figure 13.8: Comparison of model-centric and data-centric AI approaches. Model-centric AI focuses on improving architectures, while data-centric AI emphasizes enhancing dataset quality. Both approaches are complementary in optimizing AI performance.

MLOps provides the structure and practices necessary to align these specialized roles around a shared objective: delivering reliable, scalable, and maintainable machine learning systems in production environments. From designing robust data pipelines to deploying and monitoring models in live systems, effective MLOps depends on collaboration across disciplines including data engineering, statistical modeling, software development, infrastructure management, and project coordination.

13.5.1 Roles

Table 13.3 introduces the key roles that participate in MLOps and outlines their primary responsibilities. Understanding these roles not only clarifies the scope of skills required to support production ML systems but also helps frame the collaborative workflows and handoffs that drive the operational success of machine learning at scale.

Table 13.3: MLOps roles and responsibilities across the machine learning lifecycle.

Role	Primary Focus	Core Responsibilities Summary	MLOps Lifecycle Alignment
Data Engineer	Data preparation and infrastructure	Build and maintain pipelines; ensure quality, structure, and lineage of data	Data ingestion, transformation
Data Scientist	Model development and experimentation	Formulate tasks; build and evaluate models; iterate using feedback and error analysis	Modeling and evaluation
ML Engineer	Production integration and scalability	Operationalize models; implement serving logic; manage performance and retraining	Deployment and inference
DevOps Engineer	Infrastructure orchestration and automation	Manage compute infrastructure; implement CI/CD; monitor systems and workflows	Training, deployment, monitoring
Project Manager	Coordination and delivery oversight	Align goals; manage schedules and milestones; enable cross-team execution	Planning and integration
Responsible AI Lead	Ethics, fairness, and governance	Monitor bias and fairness; enforce transparency and compliance standards	Evaluation and governance
Security & Privacy Engineer	System protection and data integrity	Secure data and models; implement privacy controls; ensure system resilience	Data handling and compliance

13.5.1.1 Data Engineers

Data engineers are responsible for constructing and maintaining the data infrastructure that underpins machine learning systems. Their primary focus is to ensure that data is reliably collected, processed, and made accessible in formats suitable for analysis, feature extraction, model training, and inference. In the context of MLOps, data engineers play a foundational role by building scalable and reproducible data pipelines that support the end-to-end machine learning lifecycle.

A core responsibility of data engineers is data ingestion—extracting data from diverse operational sources such as transactional databases, web applications, log streams, and sensors. This data is typically transferred to centralized storage systems, such as cloud-based object stores (e.g., Amazon S3, Google Cloud Storage), which provide scalable and durable repositories for both raw and processed datasets. These ingestion workflows are orchestrated using scheduling and workflow tools such as Apache Airflow, Prefect, or dbt (Garg 2020).

Once ingested, the data must be transformed into structured, analysis-ready formats. This transformation process includes handling missing or malformed values, resolving inconsistencies, performing joins across heterogeneous sources, and computing derived attributes required for downstream tasks. Data engineers implement these transformations through modular pipelines that are version-controlled and designed for fault tolerance and reusability. Structured outputs are often loaded into cloud-based data warehouses such as Snowflake, Redshift, or BigQuery, or stored in feature stores for use in machine learning applications.

In addition to managing data pipelines, data engineers are responsible for provisioning and optimizing the infrastructure that supports data-intensive workflows. This includes configuring distributed storage systems, managing compute clusters, and maintaining metadata catalogs that document data schemas, lineage, and access controls. To ensure reproducibility and governance, data engineers implement dataset versioning, maintain historical snapshots, and enforce data retention and auditing policies.

For example, in a manufacturing application, data engineers may construct an Airflow pipeline that ingests time-series sensor data from programmable logic controllers (PLCs)[10] on the factory floor. The raw data is cleaned, joined with product metadata, and aggregated into statistical features such as rolling averages and thresholds. The processed features are stored in a Snowflake data warehouse, where they are consumed by downstream modeling and inference workflows.

Through their design and maintenance of robust data infrastructure, data engineers enable the consistent and efficient delivery of high-quality data. Their contributions ensure that machine learning systems are built on reliable inputs, supporting reproducibility, scalability, and operational stability across the MLOps pipeline.

To illustrate this responsibility in practice, Listing 13.1 shows a simplified example of a daily Extract-Transform-Load (ETL) pipeline implemented using Apache Airflow. This workflow automates the ingestion and transformation of raw sensor data, preparing it for downstream machine learning tasks.

13.5.1.2 Data Scientists

Data scientists are primarily responsible for designing, developing, and evaluating machine learning models. Their role centers on transforming business or operational problems into formal learning tasks, selecting appropriate algorithms, and optimizing model performance through statistical and computational techniques. Within the MLOps lifecycle, data scientists operate at the intersection of exploratory analysis and model development, contributing directly to the creation of predictive or decision-making capabilities.

The process typically begins by collaborating with stakeholders to define the problem space and establish success criteria. This includes formulating the task in machine learning terms, including classification, regression, or forecasting, and identifying suitable evaluation metrics to quantify model performance. These metrics, such as accuracy, precision, recall, area under the curve (AUC), or F1 score, provide objective measures for comparing model alternatives and guiding iterative improvements (Rainio, Teuho, and Klén 2024).

Data scientists conduct exploratory data analysis (EDA) to assess data quality, identify patterns, and uncover relationships that inform feature selection and engineering. This stage may involve statistical summaries, visualizations, and hypothesis testing to evaluate the data's suitability for modeling. Based on these findings, relevant features are constructed or selected in collaboration with data engineers to ensure consistency across development and deployment environments.

Model development involves selecting appropriate learning algorithms and constructing architectures suited to the task and data characteristics. Data scientists employ machine learning libraries such as TensorFlow, PyTorch, or scikit-learn to implement and train models. Hyperparameter tuning, regularization strategies, and cross-validation are used to optimize performance on

[10] Programmable Logic Controller (PLC): An industrial computer used to control manufacturing processes, such as robotic devices or assembly lines.

Listing 13.1: Code in Practice for a Data Engineer, implementing a daily Extract-Transform-Load (ETL) pipeline using Apache Airflow to process manufacturing sensor data.

```
# Airflow DAG for daily ETL from a manufacturing data source
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def extract_data():
    import pandas as pd
    df = pd.read_csv('/data/raw/plc_logs.csv')  # Simulated
                                                # PLC data
    df.to_parquet('/data/staged/sensor_data.parquet')

def transform_data():
    import pandas as pd
    df = pd.read_parquet('/data/staged/sensor_data.parquet')
    df['rolling_avg'] = (
      df['temperature']
      .rolling(window=10)
      .mean()
    )
    df.to_parquet('/data/processed/features.parquet')

with DAG(
    dag_id='manufacturing_etl_pipeline',
    schedule_interval='@daily',
    start_date=datetime(2023, 1, 1),
    catchup=False
) as dag:
    extract = PythonOperator(
      task_id='extract',
      python_callable=extract_data
    )
    transform = PythonOperator(
    task_id='transform',
    python_callable=transform_data
    )

    extract >> transform
```

validation datasets while mitigating overfitting. Throughout this process, tools for experiment tracking, including MLflow and Weights & Biases, are often used to log configuration settings, evaluation results, and model artifacts.

Once a candidate model demonstrates acceptable performance, it undergoes further validation through rigorous testing on holdout datasets. In addition to aggregate performance metrics, data scientists perform error analysis to identify failure modes, outliers, or biases that may impact model reliability or fairness. These insights often motivate further iterations on data processing, feature engineering, or model refinement.

Data scientists also participate in post-deployment monitoring and retraining workflows. They assist in analyzing data drift, interpreting shifts in model performance, and incorporating new data to maintain predictive accuracy over time. In collaboration with ML engineers, they define retraining strategies and evaluate the impact of updated models on operational metrics.

For example, in a retail forecasting scenario, a data scientist may develop a sequence model using TensorFlow to predict product demand based on historical sales, product attributes, and seasonal indicators. The model is evaluated using root mean squared error (RMSE) on withheld data, refined through hyperparameter tuning, and handed off to ML engineers for deployment. Following deployment, the data scientist continues to monitor model accuracy and guides retraining using new transactional data.

Through rigorous experimentation and model development, data scientists contribute the core analytical functionality of machine learning systems. Their work transforms raw data into predictive insights and supports the continuous improvement of deployed models through principled evaluation and refinement.

To illustrate these responsibilities in a practical context, Listing 13.2 presents a minimal example of a sequence model built using TensorFlow. This model is designed to forecast product demand based on historical sales patterns and other input features.

13.5.1.3 ML Engineers

Machine learning engineers are responsible for translating experimental models into reliable, scalable systems that can be integrated into real-world applications. Positioned at the intersection of data science and software engineering, ML engineers ensure that models developed in research environments can be deployed, monitored, and maintained within production-grade infrastructure. Their work bridges the gap between prototyping and operationalization, enabling machine learning to deliver sustained value in practice.

A core responsibility of ML engineers is to take trained models and encapsulate them within modular, maintainable components. This often involves refactoring code for robustness, implementing model interfaces, and building application programming interfaces (APIs) that expose model predictions to downstream systems. Frameworks such as Flask and FastAPI are commonly used to construct lightweight, RESTful services[11] for model inference. To support portability and environment consistency, models, and their dependencies are typically containerized using Docker and managed within orchestration systems like Kubernetes.

ML engineers also oversee the integration of models into continuous integration and continuous delivery (CI/CD) pipelines. These pipelines automate

[11] RESTful Services: Web services implementing REST (Representational State Transfer) principles for networked applications.

Listing 13.2: Code in Practice for a Data Scientist, implementing a sequence model using TensorFlow to forecast product demand based on historical sales data.

```
# TensorFlow model for demand forecasting
import tensorflow as tf
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Input(shape=(30, 5)),  # 30 time steps, 5 features
    layers.LSTM(64),
    layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Assume X_train, y_train are preloaded
model.fit(X_train, y_train, validation_split=0.2, epochs=10)

# Save model for handoff
model.save('models/demand_forecast_v1')
```

the retraining, testing, and deployment of models, ensuring that updated models are validated against performance benchmarks before being promoted to production. Practices such as canary deployments, A/B testing, and staged rollouts allow for gradual transitions and reduce the risk of regressions. In the event of model degradation, rollback procedures are used to restore previously validated versions.

Operational efficiency is another key area of focus. ML engineers apply a range of optimization techniques, including model quantization, pruning, and batch serving, to meet latency, throughput, and cost constraints. In systems that support multiple models, they may implement mechanisms for dynamic model selection or concurrent serving. These optimizations are closely coupled with infrastructure provisioning, which often includes the configuration of GPUs or other specialized accelerators.

Post-deployment, ML engineers play a critical role in monitoring model behavior. They configure telemetry systems to track latency, failure rates, and resource usage, and they instrument prediction pipelines with logging and alerting mechanisms. In collaboration with data scientists and DevOps engineers, they respond to changes in system behavior, trigger retraining workflows, and ensure that models continue to meet service-level objectives[12].

For example, consider a financial services application where a data science team has developed a fraud detection model using TensorFlow. An ML engineer packages the model for deployment using TensorFlow Serving, configures a REST API for integration with the transaction pipeline, and sets up a CI/CD pipeline in Jenkins to automate updates. They implement logging and monitor-

[12] Service-Level Objectives (SLOs): Specific measurable characteristics of the SLAs such as availability, throughput, frequency, response time, or quality.

ing using Prometheus and Grafana, and configure rollback logic to revert to the prior model version if performance deteriorates. This production infrastructure enables the model to operate continuously and reliably under real-world workloads.

Through their focus on software robustness, deployment automation, and operational monitoring, ML engineers play a pivotal role in transitioning machine learning models from experimental artifacts into trusted components of production systems. To illustrate these responsibilities in a practical context, Listing 13.3 presents a minimal example of a REST API built with FastAPI for serving a trained TensorFlow model. This service exposes model predictions for use in downstream applications.

Listing 13.3: Code in Practice for an ML Engineer, wrapping a trained model in a FastAPI endpoint to expose real-time demand predictions in a production environment.

```python
# FastAPI service to serve a trained TensorFlow model
from fastapi import FastAPI, Request
import tensorflow as tf
import numpy as np

app = FastAPI()
model = tf.keras.models.load_model('models/demand_forecast_v1')

@app.post("/predict")
async def predict(request: Request):
    data = await request.json()
    input_array = np.array(data['input']).reshape(1, 30, 5)
    prediction = model.predict(input_array)
    return {"prediction": float(prediction[0][0])}
```

13.5.1.4 DevOps Engineers

DevOps engineers are responsible for provisioning, managing, and automating the infrastructure that supports the development, deployment, and monitoring of machine learning systems. Originating from the broader discipline of software engineering, the role of the DevOps engineer in MLOps extends traditional responsibilities to accommodate the specific demands of data- and model-driven workflows. Their expertise in cloud computing, automation pipelines, and infrastructure as code (IaC) enables scalable and reliable machine learning operations.

A central task for DevOps engineers is the configuration and orchestration of compute infrastructure used throughout the ML lifecycle. This includes provisioning virtual machines, storage systems, and accelerators such as GPUs and TPUs using IaC tools like Terraform, AWS CloudFormation, or Ansible. Infrastructure is typically containerized using Docker and managed through orchestration platforms such as Kubernetes, which allow teams to deploy, scale, and monitor workloads across distributed environments.

DevOps engineers design and implement CI/CD pipelines tailored to machine learning workflows. These pipelines automate the retraining, testing, and deployment of models in response to code changes or data updates. Tools such as Jenkins, GitHub Actions, or GitLab CI are used to trigger model workflows, while platforms like MLflow and Kubeflow facilitate experiment tracking, model registration, and artifact versioning. By codifying deployment logic, these pipelines reduce manual effort, increase reproducibility, and enable faster iteration cycles.

Monitoring is another critical area of responsibility. DevOps engineers configure telemetry systems to collect metrics related to both model and infrastructure performance. Tools such as Prometheus, Grafana, and the ELK stack (Elasticsearch, Logstash, Kibana) are widely used to build dashboards, set thresholds, and generate alerts. These systems allow teams to detect anomalies in latency, throughput, resource utilization, or prediction behavior and respond proactively to emerging issues.

To ensure compliance and operational discipline, DevOps engineers also implement governance mechanisms that enforce consistency and traceability. This includes versioning of infrastructure configurations, automated validation of deployment artifacts, and auditing of model updates. In collaboration with ML engineers and data scientists, they enable reproducible and auditable model deployments aligned with organizational and regulatory requirements.

For instance, in a financial services application, a DevOps engineer may configure a Kubernetes cluster on AWS to support both model training and online inference. Using Terraform, the infrastructure is defined as code and versioned alongside the application repository. Jenkins is used to automate the deployment of models registered in MLflow, while Prometheus and Grafana provide real-time monitoring of API latency, resource usage, and container health.

By abstracting and automating the infrastructure that underlies ML workflows, DevOps engineers enable scalable experimentation, robust deployment, and continuous monitoring. Their role ensures that machine learning systems can operate reliably under production constraints, with minimal manual intervention and maximal operational efficiency. To illustrate these responsibilities in a practical context, Listing 13.4 presents an example of using Terraform to provision a GPU-enabled virtual machine on Google Cloud Platform for model training and inference workloads.

13.5.1.5 Project Managers

Project managers play a critical role in coordinating the activities, resources, and timelines involved in delivering machine learning systems. While they do not typically develop models or write code, project managers are essential to aligning interdisciplinary teams, tracking progress against objectives, and ensuring that MLOps initiatives are completed on schedule and within scope. Their work enables effective collaboration among data scientists, engineers, product stakeholders, and infrastructure teams, translating business goals into actionable technical plans.

Listing 13.4: Code in Practice for a DevOps Engineer, provisioning GPU-enabled infrastructure on GCP using Terraform to support model training and serving.

```
# Terraform configuration for a GCP instance with GPU support
resource "google_compute_instance" "ml_node" {
  name         = "ml-gpu-node"
  machine_type = "n1-standard-8"
  zone         = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  guest_accelerator {
    type  = "nvidia-tesla-t4"
    count = 1
  }

  metadata_startup_script = <<-EOF
    sudo apt-get update
    sudo apt-get install -y docker.io
    sudo docker run --gpus all -p 8501:8501 tensorflow/serving
  EOF

  tags = ["ml-serving"]
}
```

At the outset of a project, project managers work with organizational stakeholders to define goals, success metrics, and constraints. This includes clarifying the business objectives of the machine learning system, identifying key deliverables, estimating timelines, and setting performance benchmarks. These definitions serve as the foundation for resource allocation, task planning, and risk assessment throughout the lifecycle of the project.

Once the project is initiated, project managers are responsible for developing and maintaining a detailed execution plan. This plan outlines major phases of work, such as data collection, model development, infrastructure provisioning, deployment, and monitoring. Dependencies between tasks are identified and managed to ensure smooth handoffs between roles, while milestones and checkpoints are used to assess progress and adjust schedules as necessary.

Throughout execution, project managers facilitate coordination across teams. This includes organizing meetings, tracking deliverables, resolving blockers, and escalating issues when necessary. Documentation, progress reports, and status updates are maintained to provide visibility across the organization and ensure that all stakeholders are informed of project developments. Communication is a central function of the role, serving to reduce misalignment and clarify expectations between technical contributors and business decision-makers.

In addition to managing timelines and coordination, project managers oversee the budgeting and resourcing aspects of MLOps initiatives. This may involve evaluating cloud infrastructure costs, negotiating access to compute resources, and ensuring that appropriate personnel are assigned to each phase of the project. By maintaining visibility into both technical and organizational considerations, project managers help align technical execution with strategic priorities.

For example, consider a company seeking to reduce customer churn using a predictive model. The project manager coordinates with data engineers to define data requirements, with data scientists to prototype and evaluate models, with ML engineers to package and deploy the final model, and with DevOps engineers to provision the necessary infrastructure and monitoring tools. The project manager tracks progress through phases such as data pipeline readiness, baseline model evaluation, deployment to staging, and post-deployment monitoring, adjusting the project plan as needed to respond to emerging challenges.

By orchestrating collaboration across diverse roles and managing the complexity inherent in machine learning initiatives, project managers enable MLOps teams to deliver systems that are both technically robust and aligned with organizational goals. Their contributions ensure that the operationalization of machine learning is not only feasible, but repeatable, accountable, and efficient. To illustrate these responsibilities in a practical context, Listing 13.5 presents a simplified example of a project milestone tracking structure using JSON. This format is commonly used to integrate with tools like JIRA or project dashboards to monitor progress across machine learning initiatives.

13.5.1.6 Responsible AI Lead

The Responsible AI Lead is tasked with ensuring that machine learning systems operate in ways that are transparent, fair, accountable, and compliant with ethical and regulatory standards. As machine learning is increasingly embedded in socially impactful domains such as healthcare, finance, and education, the need for systematic governance has grown. This role reflects a growing recognition that technical performance alone is insufficient; ML systems must also align with broader societal values.

At the model development stage, Responsible AI Leads support practices that enhance interpretability and transparency. They work with data scientists and ML engineers to assess which features contribute most to model predictions, evaluate whether certain groups are disproportionately affected, and document model behavior through structured reporting mechanisms. Post hoc explanation methods, such as attribution techniques, are often reviewed in collaboration with this role to support downstream accountability.

Another key responsibility is fairness assessment. This involves defining fairness criteria in collaboration with stakeholders, auditing model outputs for performance disparities across demographic groups, and guiding interventions, including reweighting, re-labeling, or constrained optimization, to mitigate po-

Listing 13.5: Project milestone tracking in JSON

```json
{
  "project": "Churn Prediction",
  "milestones": [
    {
      "name": "Data Pipeline Ready",
      "due": "2025-05-01",
      "status": "Complete"
    },
    {
      "name": "Model Baseline",
      "due": "2025-05-10",
      "status": "In Progress"
    },
    {
      "name": "Staging Deployment",
      "due": "2025-05-15",
      "status": "Pending"
    },
    {
      "name": "Production Launch",
      "due": "2025-05-25",
      "status": "Pending"
    }
  ],
  "risks": [
    {
      "issue": "Delayed cloud quota",
      "mitigation": "Request early from infra team"
    }
  ]
}
```

tential harms. These assessments are often incorporated into model validation pipelines to ensure that they are systematically enforced before deployment.

In post-deployment settings, Responsible AI Leads help monitor systems for drift, bias amplification, and unanticipated behavior. They may also oversee the creation of documentation artifacts such as model cards or datasheets for datasets, which serve as tools for transparency and reproducibility. In regulated sectors, this role collaborates with legal and compliance teams to meet audit requirements and ensure that deployed models remain aligned with external mandates.

For example, in a hiring recommendation system, a Responsible AI Lead may oversee an audit that compares model outcomes across gender and ethnicity, guiding the team to adjust the training pipeline to reduce disparities while preserving predictive accuracy. They also ensure that decision rationales are documented and reviewable by both technical and non-technical stakeholders.

The integration of ethical review and governance into the ML development process enables the Responsible AI Lead to support systems that are not only technically robust, but also socially responsible and institutionally accountable. To illustrate these responsibilities in a practical context, Listing 13.6 presents an example of using the Aequitas library to audit a model for group-based disparities. This example evaluates statistical parity across demographic groups to assess potential fairness concerns prior to deployment.

Listing 13.6: Code in Practice for a Responsible AI Lead, conducting a fairness assessment to identify disparities in model outcomes across gender groups using Aequitas.

```python
# Fairness audit using Aequitas
from aequitas.group import Group
from aequitas.bias import Bias

# Assume df includes model scores, true labels,
# and a 'gender' attribute
g = Group().get_crosstabs(df)
b = Bias().get_disparity_predefined_groups(
    g,original_df=df,
    ref_groups_dict={'gender': 'male'},
    alpha=0.05,
    mask_significant=True
)

print(b[
    ['attribute_name',
     'attribute_value',
     'disparity',
     'statistical_parity']
])
```

13.5.1.7 Security and Privacy Engineer

The Security and Privacy Engineer is responsible for safeguarding machine learning systems against adversarial threats and privacy risks. As ML systems increasingly rely on sensitive data and are deployed in high-stakes environments, security and privacy become essential dimensions of system reliability. This role brings expertise in both traditional security engineering and ML-specific threat models, ensuring that systems are resilient to attack and compliant with data protection requirements.

At the data level, Security and Privacy Engineers help enforce access control, encryption, and secure handling of training and inference data. They collaborate with data engineers to apply privacy-preserving techniques, such as data anonymization, secure aggregation, or differential privacy[13], particularly when sensitive personal or proprietary data is used. These mechanisms are designed to reduce the risk of data leakage while retaining the utility needed for model training.

In the modeling phase, this role advises on techniques that improve robustness against adversarial manipulation. This may include detecting poisoning attacks during training, mitigating model inversion or membership inference risks, and evaluating the susceptibility of models to adversarial examples. They also assist in designing model architectures and training strategies that balance performance with safety constraints.

During deployment, Security and Privacy Engineers implement controls to protect the model itself, including endpoint hardening, API rate limiting[14], and access logging. In settings where models are exposed externally, including public-facing APIs, they may also deploy monitoring systems that detect anomalous access patterns or query-based attacks intended to extract model parameters or training data.

For instance, in a medical diagnosis system trained on patient data, a Security and Privacy Engineer might implement differential privacy during model training and enforce strict access controls on the model's inference interface. They would also validate that model explanations do not inadvertently expose sensitive information, and monitor post-deployment activity for potential misuse.

Through proactive design and continuous oversight, Security and Privacy Engineers ensure that ML systems uphold confidentiality, integrity, and availability. Their work is especially critical in domains where trust, compliance, and risk mitigation are central to system deployment and long-term operation. To illustrate these responsibilities in a practical context, Listing 13.7 presents an example of training a model using differential privacy techniques with TensorFlow Privacy. This approach helps protect sensitive information in the training data while preserving model utility.

13.5.2 Intersections and Handoffs

While each role in MLOps carries distinct responsibilities, the successful deployment and operation of machine learning systems depends on seamless collaboration across functional boundaries. Machine learning workflows are inherently interdependent, with critical handoff points connecting data acquisition, model development, system integration, and operational monitoring. Understanding these intersections is essential for designing processes that are both efficient and resilient.

One of the earliest and most critical intersections occurs between data engineers and data scientists. Data engineers construct and maintain the pipelines that ingest and transform raw data, while data scientists depend on these pipelines to access clean, structured, and well-documented datasets for analysis and modeling. Misalignment at this stage, including undocumented schema changes or inconsistent feature definitions, can lead to downstream errors that compromise model quality or reproducibility.

[13] Differential Privacy: A technique that adds randomness to dataset queries to protect individual data privacy while maintaining overall data utility.

[14] API rate limiting controls the rate at which end users can make API requests, used to protect against abuse.

Listing 13.7: Code in Practice for a Security and Privacy Engineer, applying differential privacy during model training to protect sensitive data while enabling predictive performance.

```
# Training a differentially private model with
# TensorFlow Privacy
import tensorflow as tf
from tensorflow_privacy.privacy.optimizers.dp_optimizer_keras \
    import DPKerasAdamOptimizer

# Define a simple model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(
      64,
      activation='relu',
      input_shape=(100,)
    ),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Use a DP-aware optimizer
optimizer = DPKerasAdamOptimizer(
    l2_norm_clip=1.0,
    noise_multiplier=1.1,
    num_microbatches=256,
    learning_rate=0.001
)

model.compile(
    optimizer=optimizer,
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Train model on privatized dataset
model.fit(train_data, train_labels, epochs=10, batch_size=256)
```

Once a model is developed, the handoff to ML engineers requires a careful transition from research artifacts to production-ready components. ML engineers must understand the assumptions and requirements of the model to implement appropriate interfaces, optimize runtime performance, and integrate it into the broader application ecosystem. This step often requires iteration, especially when models developed in experimental environments must be adapted to meet latency, throughput, or resource constraints in production.

As models move toward deployment, DevOps engineers play the role in provisioning infrastructure, managing CI/CD pipelines, and instrumenting monitoring systems. Their collaboration with ML engineers ensures that model deployments are automated, repeatable, and observable. They also coordinate with data scientists to define alerts and thresholds that guide performance monitoring and retraining decisions.

Project managers provide the organizational glue across these technical domains. They ensure that handoffs are anticipated, roles are clearly defined, and dependencies are actively managed. In particular, project managers help maintain continuity by documenting assumptions, tracking milestone readiness, and facilitating communication between teams. This coordination reduces friction and enables iterative development cycles that are both agile and accountable.

For example, in a real-time recommendation system, data engineers maintain the data ingestion pipeline and feature store, data scientists iterate on model architectures using historical clickstream data, ML engineers deploy models as containerized microservices, and DevOps engineers monitor inference latency and availability. Each role contributes to a different layer of the stack, but the overall functionality depends on reliable transitions between each phase of the lifecycle.

These role interactions illustrate that MLOps is not simply a collection of discrete tasks, but a continuous, collaborative process. Designing for clear handoffs, shared tools, and well-defined interfaces is essential for ensuring that machine learning systems can evolve, scale, and perform reliably over time.

13.5.3 Evolving Roles and Specializations

As machine learning systems mature and organizations adopt MLOps practices at scale, the structure and specialization of roles often evolve. In early-stage environments, individual contributors may take on multiple responsibilities—such as a data scientist who also builds data pipelines or manages model deployment. However, as systems grow in complexity and teams expand, responsibilities tend to become more differentiated, giving rise to new roles and more structured organizational patterns.

One emerging trend is the formation of dedicated ML platform teams, which focus on building shared infrastructure and tooling to support experimentation, deployment, and monitoring across multiple projects. These teams often abstract common workflows, including data versioning, model training orchestration, and CI/CD integration, into reusable components or internal platforms. This approach reduces duplication of effort and accelerates development by enabling application teams to focus on domain-specific problems rather than underlying systems engineering.

In parallel, hybrid roles have emerged to bridge gaps between traditional boundaries. For example, full-stack ML engineers[15] combine expertise in modeling, software engineering, and infrastructure to own the end-to-end deployment of ML models. Similarly, ML enablement roles, including MLOps engineers and applied ML specialists, focus on helping teams adopt best practices, integrate tooling, and scale workflows efficiently. These roles are especially valuable in organizations with diverse teams that vary in ML maturity or technical specialization.

[15] Full-stack ML engineer: A role that encompasses the skills of machine learning, software development, and system operations to handle end-to-end machine learning model lifecycle.

The structure of MLOps teams also varies based on organizational scale, industry, and regulatory requirements. In smaller organizations or startups, teams are often lean and cross-functional, with close collaboration and informal processes. In contrast, larger enterprises may formalize roles and introduce governance frameworks to manage compliance, data security, and model risk. Highly regulated sectors, including finance, healthcare, and defense, often require additional roles focused on validation, auditing, and documentation to meet external reporting obligations.

Table 13.4: Evolution of MLOps roles and responsibilities.

Role	Key Intersections	Evolving Patterns and Specializations
Data Engineer	Works with data scientists to define features and pipelines	Expands into real-time data systems and feature store platforms
Data Scientist	Relies on data engineers for clean inputs; collaborates with ML engineers	Takes on model validation, interpretability, and ethical considerations
ML Engineer	Receives models from data scientists; works with DevOps to deploy and monitor	Transitions into platform engineering or full-stack ML roles
DevOps Engineer	Supports ML engineers with infrastructure, CI/CD, and observability	Evolves into MLOps platform roles; integrates governance and security tooling
Project Manager	Coordinates across all roles; tracks progress and communication	Specializes into ML product management as systems scale
Responsible AI Lead	Collaborates with data scientists and PMs to evaluate fairness and compliance	Role emerges as systems face regulatory scrutiny or public exposure
Security & Privacy Engineer	Works with DevOps and ML Engineers to secure data pipelines and model interfaces	Role formalizes as privacy regulations (e.g., GDPR, HIPAA) apply to ML workflows

Importantly, as Table 13.4 indicates, the boundaries between roles are not rigid. Effective MLOps practices rely on shared understanding, documentation, and tools that facilitate communication and coordination across teams. Encouraging interdisciplinary fluency, including enabling data scientists to understand deployment workflows and DevOps engineers to interpret model monitoring metrics, enhances organizational agility and resilience.

As machine learning becomes increasingly central to modern software systems, roles will continue to adapt in response to emerging tools, methodologies, and system architectures. Recognizing the dynamic nature of these responsibilities allows teams to allocate resources effectively, design adaptable workflows, and foster collaboration that is essential for sustained success in production-scale machine learning.

13.6 Operational System Design

Machine learning systems do not operate in isolation. As they transition from prototype to production, their effectiveness depends not only on the quality of the underlying models, but also on the maturity of the organizational and technical processes that support them. Operational maturity refers to the degree to which ML workflows are automated, reproducible, monitored, and aligned with broader engineering and governance practices. While early-stage efforts may rely on ad hoc scripts and manual interventions, production-scale systems require deliberate design choices that support long-term sustainability, reliability, and adaptability. This section examines how different levels of operational maturity influence system architecture, infrastructure design, and organizational structure, providing a lens through which to interpret the broader MLOps landscape (Kreuzberger, Kerschbaum, and Kuhn 2022).

13.6.1 Operational Maturity

Operational maturity in machine learning refers to the extent to which an organization can reliably develop, deploy, and manage ML systems in a repeatable and scalable manner. Unlike the maturity of individual models or algorithms, operational maturity reflects systemic capabilities: how well a team or organization integrates infrastructure, automation, monitoring, governance, and collaboration into the ML lifecycle.

Low-maturity environments often rely on manual workflows, loosely coupled components, and ad hoc experimentation. While sufficient for early-stage research or low-risk applications, such systems tend to be brittle, difficult to reproduce, and highly sensitive to data or code changes. As ML systems are deployed at scale, these limitations quickly become barriers to sustained performance, trust, and accountability.

In contrast, high-maturity environments implement modular, versioned, and automated workflows that allow models to be developed, validated, and deployed in a controlled and observable fashion. Data lineage is preserved across transformations; model behavior is continuously monitored and evaluated; and infrastructure is provisioned and managed as code. These practices reduce operational friction, enable faster iteration, and support robust decision-making in production (Zaharia et al. 2018).

Importantly, operational maturity is not solely a function of tool adoption. While technologies such as CI/CD pipelines, model registries, and observability stacks play a role, maturity is fundamentally about system integration and coordination, as in how these components work together to support reliability, reproducibility, and responsiveness under real-world constraints. It is this integration that distinguishes mature ML systems from collections of loosely connected artifacts.

13.6.2 Maturity Levels

While operational maturity exists on a continuum, it is useful to distinguish between broad stages that reflect how ML systems evolve from research prototypes to production-grade infrastructure. These stages are not strict categories, but rather indicative of how organizations gradually adopt practices that support reliability, scalability, and observability.

At the lowest level of maturity, ML workflows are ad hoc: experiments are run manually, models are trained on local machines, and deployment involves hand-crafted scripts or manual intervention. Data pipelines may be fragile or undocumented, and there is limited ability to trace how a deployed model was produced. These environments may be sufficient for prototyping, but they are ill-suited for ongoing maintenance or collaboration.

As maturity increases, workflows become more structured and repeatable. Teams begin to adopt version control, automated training pipelines, and centralized model storage. Monitoring and testing frameworks are introduced, and retraining workflows become more systematic. Systems at this level can support limited scale and iteration but still rely heavily on human coordination.

At the highest levels of maturity, ML systems are fully integrated with infrastructure-as-code, continuous delivery pipelines, and automated monitoring. Data lineage, feature reuse, and model validation are encoded into the development process. Governance is embedded throughout the system, allowing for traceability, auditing, and policy enforcement. These environments support large-scale deployment, rapid experimentation, and adaptation to changing data and system conditions.

This progression, summarized in Table 13.5, offers a system-level framework for analyzing ML operational practices. It emphasizes architectural cohesion and lifecycle integration over tool selection, guiding the design of scalable and maintainable learning systems.

Table 13.5: Maturity levels in machine learning operations.

Maturity Level	System Characteristics	Typical Outcomes
Ad Hoc	Manual data processing, local training, no version control, unclear ownership	Fragile workflows, difficult to reproduce or debug
Repeatable	Automated training pipelines, basic CI/CD, centralized model storage, some monitoring	Improved reproducibility, limited scalability
Scalable	Fully automated workflows, integrated observability, infrastructure-as-code, governance	High reliability, rapid iteration, production-grade ML

These maturity levels provide a systems lens through which to evaluate ML operations—not in terms of specific tools adopted, but in how reliably and cohesively a system supports the full machine learning lifecycle. Understanding this progression prepares practitioners to identify design bottlenecks and prioritize investments that support long-term system sustainability.

13.6.3 System Design Implications

As machine learning operations mature, the underlying system architecture evolves in response. Operational maturity is not just an organizational concern—it has direct consequences for how ML systems are structured, deployed, and maintained. Each level of maturity introduces new expectations around modularity, automation, monitoring, and fault tolerance, shaping the design space in both technical and procedural terms.

In low-maturity environments, ML systems are often constructed around monolithic scripts and tightly coupled components. Data processing logic may be embedded directly within model code, and configurations are managed informally. These architectures, while expedient for rapid experimentation, lack the separation of concerns needed for maintainability, version control, or safe iteration. As a result, teams frequently encounter regressions, silent failures, and inconsistent performance across environments.

As maturity increases, modular abstractions begin to emerge. Feature engineering is decoupled from model logic, pipelines are defined declaratively, and system boundaries are enforced through APIs and orchestration frameworks. These changes support reproducibility and enable teams to scale development across multiple contributors or applications. Infrastructure becomes programmable through configuration files, and model artifacts are promoted through standardized deployment stages. This architectural discipline allows systems to evolve predictably, even as requirements shift or data distributions change.

At high levels of maturity, ML systems exhibit properties commonly found in production-grade software systems: stateless services, contract-driven interfaces, environment isolation, and observable execution. Design patterns such as feature stores, model registries, and infrastructure-as-code become foundational. Crucially, system behavior is not inferred from static assumptions, but monitored in real time and adapted as needed. This enables feedback-driven development and supports closed-loop systems where data, models, and infrastructure co-evolve.

In each case, operational maturity is not an external constraint but an architectural force: it governs how complexity is managed, how change is absorbed, and how the system can scale in the face of threats to service uptime (see Figure 13.9). Design decisions that disregard these constraints may function under ideal conditions, but fail under real-world pressures such as latency requirements, drift, outages, or regulatory audits. Understanding this relationship between maturity and design is essential for building resilient machine learning systems that sustain performance over time.

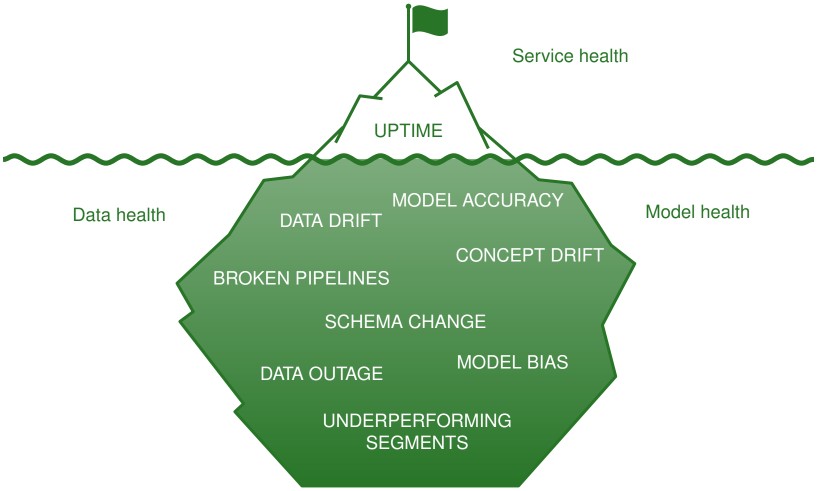


Figure 13.9: How ML service uptime is supported by an "iceberg" of underlying components to monitor.

13.6.4 Patterns and Anti-Patterns

The structure of the teams involved in building and maintaining machine learning systems plays a significant role in determining operational outcomes. As ML systems grow in complexity and scale, organizational patterns must evolve to reflect the interdependence between data, modeling, infrastructure, and governance. While there is no single ideal structure, certain patterns consistently support operational maturity, whereas others tend to hinder it.

In mature environments, organizational design emphasizes clear ownership, cross-functional collaboration, and interface discipline between roles. For instance, platform teams may take responsibility for shared infrastructure, tooling, and CI/CD pipelines, while domain teams focus on model development and business alignment. This separation of concerns enables reuse, standardization, and parallel development. Interfaces between teams, including feature definitions, data schemas, and deployment targets, are well-defined and versioned, reducing friction and ambiguity.

One effective pattern is the creation of a centralized MLOps team that provides shared services to multiple model development groups. This team maintains tooling for model training, validation, deployment, and monitoring, and may operate as an internal platform provider. Such structures promote consistency, reduce duplicated effort, and accelerate onboarding for new projects. Alternatively, some organizations adopt a federated model, embedding MLOps engineers within product teams while maintaining a central architectural function to guide system-wide integration.

In contrast, anti-patterns often emerge when responsibilities are fragmented or poorly aligned. One common failure mode is the tool-first approach, in which teams adopt infrastructure or automation tools without first defining the processes and roles that should govern their use. This can result in fragile pipelines, unclear handoffs, and duplicated effort. Another anti-pattern is siloed experimentation, where data scientists operate in isolation from production engineers, leading to models that are difficult to deploy, monitor, or retrain effectively.

Organizational drift is another subtle challenge. As teams scale, undocumented workflows and informal agreements may become entrenched, increasing the cost of coordination and reducing transparency. Without deliberate system design and process review, even previously functional structures can accumulate technical and organizational debt.

Ultimately, organizational maturity must co-evolve with system complexity. Teams must establish communication patterns, role definitions, and accountability structures that reinforce the principles of modularity, automation, and observability. Operational excellence in machine learning is not just a matter of technical capability—it is the product of coordinated, intentional systems thinking across human and computational boundaries.

13.6.5 Contextualizing MLOps

The operational maturity of a machine learning system is not an abstract ideal; it is realized in concrete systems with physical, organizational, and regulatory constraints. While the preceding sections have outlined best practices for mature MLOps, which include CI/CD, monitoring, infrastructure provisioning, and governance, these practices are rarely deployed in pristine, unconstrained environments. In reality, every ML system operates within a specific context that shapes how MLOps workflows are implemented, prioritized, and adapted.

System constraints may arise from the physical environment in which a model is deployed, such as limitations in compute, memory, or power. These are common in edge and embedded systems, where models must run under strict latency and resource constraints. Connectivity limitations, such as intermittent network access or bandwidth caps, further complicate model updates, monitoring, and telemetry collection. In high-assurance domains, including healthcare, finance, and industrial control systems, governance, traceability, and fail-safety may take precedence over throughput or latency. These factors do not simply influence system performance; they fundamentally alter how MLOps pipelines must be designed and maintained.

For instance, a standard CI/CD pipeline for retraining and deployment may be infeasible in environments where direct access to the model host is not possible. In such cases, teams must implement alternative delivery mechanisms, such as over-the-air updates, that account for reliability, rollback capability, and compatibility across heterogeneous devices. Similarly, monitoring practices that assume full visibility into runtime behavior may need to be reimagined using indirect signals, coarse-grained telemetry, or on-device anomaly detection. Even the simple task of collecting training data may be limited by privacy concerns, device-level storage constraints, or legal restrictions on data movement.

These adaptations should not be interpreted as deviations from maturity, but rather as expressions of maturity under constraint. A well-engineered ML system accounts for the realities of its operating environment and revises its operational practices accordingly. This is the essence of systems thinking in MLOps: applying general principles while designing for specificity.

As we turn to the chapters ahead, we will encounter several of these contextual factors, including on-device learning, privacy preservation, safety and robustness, and sustainability. Each presents not just a technical challenge but a system-level constraint that reshapes how machine learning is practiced and maintained at scale. Understanding MLOps in context is therefore not optional—it is foundational to building ML systems that are viable, trustworthy, and effective in the real world.

13.6.6 Looking Ahead

As this chapter has shown, the deployment and maintenance of machine learning systems require more than technical correctness at the model level. They demand architectural coherence,[16] organizational alignment, and operational maturity. The progression from ad hoc experimentation to scalable, auditable systems reflects a broader shift: machine learning is no longer confined to research environments—it is a core component of production infrastructure.

Understanding the maturity of an ML system helps clarify what challenges are likely to emerge and what forms of investment are needed to address them. Early-stage systems benefit from process discipline and modular abstraction; mature systems require automation, governance, and resilience. Design choices made at each stage influence the pace of experimentation, the robustness of deployed models, and the ability to integrate evolving requirements—technical, organizational, and regulatory.

This systems-oriented view of MLOps also sets the stage for the next phase of this book. The remaining chapters examine specific application contexts and operational concerns, including on-device inference, privacy, robustness, and sustainability, that depend on the foundational capabilities developed in this

[16] Refers to the logical, consistent, and scalable design and integration of various system components.

chapter. These topics represent not merely extensions of model performance, but domains in which operational maturity directly enables feasibility, safety, and long-term value.

Operational maturity is therefore not the end of the machine learning system lifecycle—it is the foundation upon which production-grade, responsible, and adaptive systems are built. The following chapters explore what it takes to build such systems under domain-specific constraints, further expanding the scope of what it means to engineer machine learning at scale.

13.7 Case Studies

To ground the principles of MLOps in practice, we examine two illustrative case studies that demonstrate the operationalization of machine learning in real-world systems. These examples highlight how operational maturity, robust system design, and cross-functional collaboration enable the successful deployment and maintenance of ML applications.

The first case study analyzes the Oura Ring, which is a consumer wearable device that employs embedded machine learning to monitor sleep and physiological signals. This example illustrates MLOps practices in resource-constrained environments, where models must operate efficiently on edge devices while maintaining reliability and accuracy. The second case study explores ClinAIOps, a specialized framework for deploying AI systems in clinical settings. By examining its application to continuous therapeutic monitoring (CTM), we see how MLOps principles can be adapted to domains with strict regulatory requirements and complex human-in-the-loop workflows.

Through these cases, we gain practical insights into how organizations navigate technical, operational, and domain-specific challenges in productionizing machine learning systems. Each example reinforces core MLOps concepts while revealing unique considerations that arise in different application contexts.

13.7.1 Oura Ring Case Study

13.7.1.1 Context and Motivation

The Oura Ring is a consumer-grade wearable device designed to monitor sleep, activity, and physiological recovery through embedded sensing and computation. By measuring signals such as motion, heart rate, and body temperature, the device estimates sleep stages and delivers personalized feedback to users. Unlike traditional cloud-based systems, much of the Oura Ring's data processing and inference occurs directly on the device, making it a practical example of embedded machine learning in production.

The central objective for the development team was to improve the device's accuracy in classifying sleep stages, aligning its predictions more closely with those obtained through polysomnography (PSG)[17]—the clinical gold standard for sleep monitoring. Initial evaluations revealed a 62% correlation between the Oura Ring's predictions and PSG-derived labels, in contrast to the 82–83% correlation observed between expert human scorers. This discrepancy highlighted both the promise and limitations of the initial model, prompting a systematic effort to re-evaluate data collection, preprocessing, and model development

[17] Polysomnography (PSG): A clinical study or test used to diagnose sleep disorders; involves EEG and other physiologic sensors.

workflows. As such, the case illustrates the importance of robust MLOps practices, particularly when operating under the constraints of embedded systems.

13.7.1.2 Data Acquisition and Preprocessing

To overcome the performance limitations of the initial model, the Oura team focused on constructing a robust, diverse dataset grounded in clinical standards. They designed a large-scale sleep study involving 106 participants from three continents, including Asia, Europe, and North America, capturing broad demographic variability across age, gender, and lifestyle. During the study, each participant wore the Oura Ring while simultaneously undergoing polysomnography (PSG), the clinical gold standard for sleep staging. This pairing enabled the creation of a high-fidelity labeled dataset aligning wearable sensor data with validated sleep annotations.

In total, the study yielded 440 nights of data and over 3,400 hours of time-synchronized recordings. This dataset captured not only physiological diversity but also variability in environmental and behavioral factors, which is critical for generalizing model performance across a real-world user base.

To manage the complexity and scale of this dataset, the team implemented automated data pipelines for ingestion, cleaning, and preprocessing. Physiological signals, comprising heart rate, motion, and body temperature, were extracted and validated using structured workflows. Leveraging the Edge Impulse platform[18], they consolidated raw inputs from multiple sources, resolved temporal misalignments, and structured the data for downstream model development. These workflows significantly reduced the need for manual intervention, highlighting how MLOps principles such as pipeline automation, data versioning, and reproducible preprocessing are essential in embedded ML settings.

13.7.2 Model Development and Evaluation

With a high-quality, clinically labeled dataset in place, the Oura team advanced to the development and evaluation of machine learning models designed to classify sleep stages. Recognizing the operational constraints of wearable devices, model design prioritized efficiency and interpretability alongside predictive accuracy. Rather than employing complex architectures typical of server-scale deployments, the team selected models that could operate within the ring's limited memory and compute budget.

Two model configurations were explored. The first used only accelerometer data, representing a lightweight architecture optimized for minimal energy consumption and low-latency inference. The second model incorporated additional physiological inputs, including heart rate variability and body temperature, enabling the capture of autonomic nervous system activity and circadian rhythms—factors known to correlate with sleep stage transitions.

To evaluate performance, the team applied five-fold cross-validation and benchmarked the models against the gold-standard PSG annotations. Through iterative tuning of hyperparameters and refinement of input features, the enhanced models achieved a correlation accuracy of 79%, significantly surpassing

[18] Edge Impulse: A development platform for embedded machine learning, enabling data collection, model training, and deployment on edge devices.

the original system's 62% correlation and approaching the clinical benchmark of 82–83%.

These performance gains did not result solely from architectural innovation. Instead, they reflect the broader impact of a systematic MLOps approach—one that integrated rigorous data collection, reproducible training pipelines, and disciplined evaluation practices. This phase underscores the importance of aligning model development with both application constraints and system-level reliability, particularly in embedded ML environments where deployment feasibility is as critical as accuracy.

13.7.3 Deployment and Iteration

Following model validation, the Oura team transitioned to deploying the trained models onto the ring's embedded hardware. Deployment in this context required careful accommodation of strict constraints on memory, compute, and power. The lightweight model, which relied solely on accelerometer input, was particularly well-suited for real-time inference on-device, delivering low-latency predictions with minimal energy usage. In contrast, the more comprehensive model, which utilized additional physiological signals, including heart rate variability and temperature, was deployed selectively, where higher predictive fidelity was required and system resources permitted.

To facilitate reliable and scalable deployment, the team developed a modular toolchain for converting trained models into optimized formats suitable for embedded execution. This process included model compression techniques such as quantization and pruning, which reduced model size while preserving accuracy. Models were packaged with their preprocessing routines and deployed using over-the-air (OTA) update mechanisms, ensuring consistency across devices in the field.

Instrumentation was built into the deployment pipeline to support post-deployment observability. The system collected operational telemetry, including runtime performance metrics, device-specific conditions, and samples of model predictions. This monitoring infrastructure enabled the identification of drift, edge cases, and emerging patterns in real-world usage, closing the feedback loop between deployment and further development.

This stage illustrates key practices of MLOps in embedded systems: resource-aware model packaging, OTA deployment infrastructure, and continuous performance monitoring. It reinforces the importance of designing systems for adaptability and iteration, ensuring that ML models remain accurate and reliable under real-world operating conditions.

13.7.4 Lessons from MLOps Practice

The Oura Ring case study illustrates several essential principles for managing machine learning systems in real-world, resource-constrained environments. First, it highlights the foundational role of data quality and labeling. While model architecture and training pipelines are important, the success of the system was driven by a disciplined approach to data acquisition, annotation, and preprocessing. This affirms the importance of data-centric practices in MLOps workflows.

Second, the deployment strategy demonstrates the need for system-aware model design. Rather than relying on a single large model, the team developed tiered models optimized for different deployment contexts. This modularity enabled tradeoffs between accuracy and efficiency to be managed at runtime, a key consideration for on-device and embedded inference.

Third, the case emphasizes the value of operational feedback loops. Instrumentation for logging and monitoring allowed the team to track system behavior post-deployment, identify shortcomings, and guide further iterations. This reinforces the role of observability and feedback as core components of the MLOps lifecycle.

Finally, the success of the Oura project was not due to a single team or phase of work but emerged from coordinated collaboration across data engineers, ML researchers, embedded systems developers, and operations personnel. The ability to move seamlessly from data acquisition to deployment reflects the maturity of the MLOps practices involved.

Taken together, this case exemplifies how MLOps is not merely a set of tools or techniques but a mindset for integrating ML into end-to-end systems that are reliable, scalable, and adaptive in production settings.

13.7.5 ClinAIOps Case Study

The deployment of machine learning systems in healthcare presents both a significant opportunity and a unique challenge. While traditional MLOps frameworks offer structured practices for managing model development, deployment, and monitoring, they often fall short in domains that require extensive human oversight, domain-specific evaluation, and ethical governance. Medical health monitoring, especially through continuous therapeutic monitoring (CTM), is one such domain where MLOps must evolve to meet the demands of real-world clinical integration.

CTM leverages wearable sensors and devices to collect rich streams of physiological and behavioral data from patients in real time. These data streams offer clinicians the potential to tailor treatments more dynamically, shifting from reactive care to proactive, personalized interventions. Recent advances in embedded ML have made this increasingly feasible. For example, wearable biosensors can automate insulin dosing for diabetes management (Psoma and Kanthou 2023), ECG-equipped wristbands can inform blood thinner adjustments for atrial fibrillation (Attia et al. 2018; Guo et al. 2019), and gait-monitoring accelerometers can trigger early interventions to prevent mobility decline in older adults (Yingcheng Liu et al. 2022). By closing the loop between sensing and therapeutic response, CTM systems powered by embedded ML are redefining how care is delivered beyond the clinical setting.

However, the mere deployment of ML models is insufficient to realize these benefits. AI systems must be integrated into clinical workflows, aligned with regulatory requirements, and designed to augment rather than replace human decision-making. The traditional MLOps paradigm, which focuses on automating pipelines for model development and serving, does not adequately account for the complex sociotechnical landscape of healthcare, where patient safety, clinician judgment, and ethical constraints must be prioritized.

This case study explores ClinAIOps, a framework proposed for operationalizing AI in clinical environments (E. Chen et al. 2023). Unlike conventional MLOps, ClinAIOps introduces mechanisms for multi-stakeholder coordination through structured feedback loops that connect patients, clinicians, and AI systems. The framework is designed to facilitate adaptive decision-making, ensure transparency and oversight, and support continuous improvement of both models and care protocols.

Before presenting a real-world application example, it is helpful to examine the limitations of traditional MLOps in clinical settings:

- MLOps focuses primarily on the model lifecycle (e.g., training, deployment, monitoring), whereas healthcare requires coordination among diverse human actors, such as patients, clinicians, and care teams.
- Traditional MLOps emphasizes automation and system reliability, but clinical decision-making hinges on personalized care, interpretability, and shared accountability.
- The ethical, regulatory, and safety implications of AI-driven healthcare demand governance frameworks that go beyond technical monitoring.
- Clinical validation requires not just performance metrics but evidence of safety, efficacy, and alignment with care standards.
- Health data is highly sensitive, and systems must comply with strict privacy and security regulations—considerations that traditional MLOps frameworks do not fully address.

In light of these gaps, ClinAIOps presents an alternative: a framework for embedding ML into healthcare in a way that balances technical rigor with clinical utility, operational reliability with ethical responsibility. The remainder of this case study introduces the ClinAIOps framework and its feedback loops, followed by a detailed walkthrough of a hypertension management example that illustrates how AI can be effectively integrated into routine clinical practice.

13.7.5.1 Feedback Loops

At the core of the ClinAIOps framework are three interlocking feedback loops that enable the safe, effective, and adaptive integration of machine learning into clinical practice. As illustrated in Figure 13.10, these loops are designed to coordinate inputs from patients, clinicians, and AI systems, facilitating data-driven decision-making while preserving human accountability and clinical oversight.

In this model, the patient is central—contributing real-world physiological data, reporting outcomes, and serving as the primary beneficiary of optimized care. The clinician interprets this data in context, provides clinical judgment, and oversees treatment adjustments. Meanwhile, the AI system continuously analyzes incoming signals, surfaces actionable insights, and learns from feedback to improve its recommendations.

Each feedback loop plays a distinct yet interconnected role:

- The **Patient-AI loop** captures and interprets real-time physiological data, generating tailored treatment suggestions.

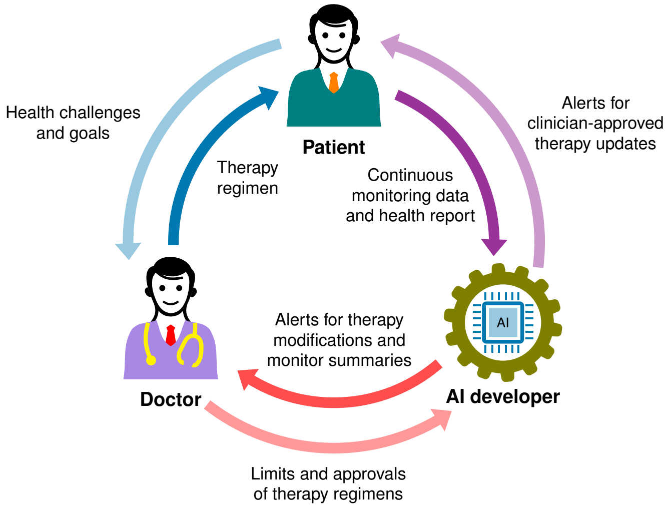

Patient
Doctor
AI developer
AI
Health challenges and goals
Therapy regimen
Continuous monitoring data and health report
Alerts for clinician-approved therapy updates
Alerts for therapy modifications and monitor summaries
Limits and approvals of therapy regimens

Figure 13.10: ClinAIOps cycle. Source: E. Chen et al. (2023).

- The **Clinician-AI loop** ensures that AI-generated recommendations are reviewed, vetted, and refined under professional supervision.
- The **Patient-Clinician loop** supports shared decision-making, empowering patients and clinicians to collaboratively set goals and interpret data trends.

Together, these loops enable adaptive personalization of care. They help calibrate AI system behavior to the evolving needs of each patient, maintain clinician control over treatment decisions, and promote continuous model improvement based on real-world feedback. By embedding AI within these structured interactions, instead of isolating it as a standalone tool, ClinAIOps provides a blueprint for responsible and effective AI integration into clinical workflows.

Patient-AI Loop. The patient–AI loop enables personalized and timely therapy optimization by leveraging continuous physiological data collected through wearable devices. Patients are equipped with sensors such as smartwatches, skin patches, or specialized biosensors that passively capture health-related signals in real-world conditions. For instance, a patient managing diabetes may wear a continuous glucose monitor, while individuals with cardiovascular conditions may use ECG-enabled wearables[19] to track cardiac rhythms.

The AI system continuously analyzes these data streams in conjunction with relevant clinical context drawn from the patient's electronic medical records, including diagnoses, lab values, prescribed medications, and demographic information. Using this holistic view, the AI model generates individualized recommendations for treatment adjustments—such as modifying dosage levels, altering administration timing, or flagging anomalous trends for review.

To ensure both responsiveness and safety, treatment suggestions are tiered. Minor adjustments that fall within clinician-defined safety thresholds may be acted upon directly by the patient, empowering self-management while

[19] Electrocardiogram (ECG): A test that records the electrical activity of the heart over a period of time using electrodes placed on the skin.

reducing clinical burden. More significant changes require review and approval by a healthcare provider. This structure maintains human oversight while enabling high-frequency, data-driven adaptation of therapies.

By enabling real-time, tailored interventions, including automatic insulin dosing adjustments based on glucose trends, this loop exemplifies how machine learning can close the feedback gap between sensing and treatment, allowing for dynamic, context-aware care outside of traditional clinical settings.

Clinician-AI Loop. The clinician–AI loop introduces a critical layer of human oversight into the process of AI-assisted therapeutic decision-making. In this loop, the AI system generates treatment recommendations and presents them to the clinician along with concise, interpretable summaries of the underlying patient data. These summaries may include longitudinal trends, sensor-derived metrics, and contextual factors extracted from the electronic health record[20].

For example, an AI model might recommend a reduction in antihypertensive medication dosage for a patient whose blood pressure has remained consistently below target thresholds. The clinician reviews the recommendation in the context of the patient's broader clinical profile and may choose to accept, reject, or modify the proposed change. This feedback, in turn, contributes to the continuous refinement of the model, improving its alignment with clinical practice.

Crucially, clinicians also define the operational boundaries within which the AI system can autonomously issue recommendations. These constraints ensure that only low-risk adjustments are automated, while more significant decisions require human approval. This preserves clinical accountability, supports patient safety, and enhances trust in AI-supported workflows.

The clinician–AI loop exemplifies a hybrid model of care in which AI augments rather than replaces human expertise. By enabling efficient review and oversight of algorithmic outputs, it facilitates the integration of machine intelligence into clinical practice while preserving the role of the clinician as the final decision-maker.

Patient-Clinician Loop. The patient–clinician loop enhances the quality of clinical interactions by shifting the focus from routine data collection to higher-level interpretation and shared decision-making. With AI systems handling data aggregation and basic trend analysis, clinicians are freed to engage more meaningfully with patients—reviewing patterns, contextualizing insights, and setting personalized health goals.

For example, in managing diabetes, a clinician may use AI-summarized data to guide a discussion on dietary habits and physical activity, tailoring recommendations to the patient's specific glycemic trends. Rather than adhering to fixed follow-up intervals, visit frequency can be adjusted dynamically based on patient progress and stability, ensuring that care delivery remains responsive and efficient.

This feedback loop positions the clinician not merely as a prescriber but as a coach and advisor, interpreting data through the lens of patient preferences, lifestyle, and clinical judgment. It reinforces the therapeutic alliance[21] by fostering collaboration and mutual understanding—key elements in personalized and patient-centered care.

[20] Electronic Health Record (EHR): A digital system that stores patient health information, used across treatment settings.

[21] The partnership formed between a clinician and a patient that enhances treatment effectiveness.

13.7.5.2 Hypertension Case Example

To concretize the principles of ClinAIOps, consider the management of hypertension—a condition affecting nearly half of adults in the United States (48.1%, or approximately 119.9 million individuals, according to the Centers for Disease Control and Prevention). Effective hypertension control often requires individualized, ongoing adjustments to therapy, making it an ideal candidate for continuous therapeutic monitoring.

ClinAIOps offers a structured framework for managing hypertension by integrating wearable sensing technologies, AI-driven recommendations, and clinician oversight into a cohesive feedback system. In this context, wearable devices equipped with photoplethysmography (PPG) and electrocardiography (ECG) sensors passively capture cardiovascular data, which can be analyzed in near-real-time to inform treatment adjustments. These inputs are augmented by behavioral data (e.g., physical activity) and medication adherence logs, forming the basis for an adaptive and responsive treatment regimen.

The following subsections detail how the patient–AI, clinician–AI, and patient–clinician loops apply in this setting, illustrating the practical implementation of ClinAIOps for a widespread and clinically significant condition.

Data Collection. In a ClinAIOps-based hypertension management system, data collection is centered on continuous, multimodal physiological monitoring. Wrist-worn devices equipped with photoplethysmography (PPG) and electrocardiography (ECG) sensors provide noninvasive estimates of blood pressure (Q. Zhang, Zhou, and Zeng 2017). These wearables also include accelerometers to capture physical activity patterns, enabling contextual interpretation of blood pressure fluctuations in relation to movement and exertion.

Complementary data inputs include self-reported logs of antihypertensive medication intake, specifying dosage and timing, as well as demographic attributes and clinical history extracted from the patient's electronic health record. Together, these heterogeneous data streams form a rich, temporally aligned dataset that captures both physiological states and behavioral factors influencing blood pressure regulation.

By integrating real-world sensor data with longitudinal clinical information, this comprehensive data foundation enables the development of personalized, context-aware models for adaptive hypertension management.

AI Model. The AI component in a ClinAIOps-driven hypertension management system is designed to operate directly on the device or in close proximity to the patient, enabling near real-time analysis and decision support. The model ingests continuous streams of blood pressure estimates, circadian rhythm indicators, physical activity levels, and medication adherence patterns to generate individualized therapeutic recommendations.

Using machine learning techniques, the model infers optimal medication dosing and timing strategies to maintain target blood pressure levels. Minor dosage adjustments that fall within predefined safety thresholds can be communicated directly to the patient, while recommendations involving more substantial modifications are routed to the supervising clinician for review and approval.

Importantly, the model supports continual refinement through a feedback mechanism that incorporates clinician decisions and patient outcomes. By integrating this observational data into subsequent training iterations, the system incrementally improves its predictive accuracy and clinical utility. The overarching objective is to enable fully personalized, adaptive blood pressure management that evolves in response to each patient's physiological and behavioral profile.

Patient-AI Loop. The patient-AI loop facilitates timely, personalized medication adjustments by delivering AI-generated recommendations directly to the patient through a wearable device or associated mobile application. When the model identifies a minor dosage modification that falls within a pre-approved safety envelope, the patient may act on the suggestion independently, enabling a form of autonomous, yet bounded, therapeutic self-management.

For recommendations involving significant changes to the prescribed regimen, the system defers to clinician oversight, ensuring medical accountability and compliance with regulatory standards. This loop empowers patients to engage actively in their care while maintaining a safeguard for clinical appropriateness.

By enabling personalized, data-driven feedback on a daily basis, the patient-AI loop supports improved adherence and therapeutic outcomes. It operationalizes a key principle of ClinAIOps, by closing the loop between continuous monitoring and adaptive intervention, while preserving the patient's role as an active agent in the treatment process.

Clinician-AI Loop. The clinician-AI loop ensures medical oversight by placing healthcare providers at the center of the decision-making process. Clinicians receive structured summaries of the patient's longitudinal blood pressure patterns, visualizations of adherence behaviors, and relevant contextual data aggregated from wearable sensors and electronic health records. These insights support efficient and informed review of the AI system's recommended medication adjustments.

Before reaching the patient, the clinician evaluates each proposed dosage change, choosing to approve, modify, or reject the recommendation based on their professional judgment and understanding of the patient's broader clinical profile. Furthermore, clinicians define the operational boundaries within which the AI may act autonomously, specifying thresholds for dosage changes that can be enacted without direct review.

When the system detects blood pressure trends indicative of clinical risk, including persistent hypotension or a hypertensive crisis[22], it generates alerts for immediate clinician intervention. These capabilities preserve the clinician's authority over treatment while enhancing their ability to manage patient care proactively and at scale.

This loop exemplifies the principles of accountability, safety, and human-in-the-loop[23] governance, ensuring that AI functions as a supportive tool rather than an autonomous agent in therapeutic decision-making.

Patient-Clinician Loop. As illustrated in Figure 13.11, the patient-clinician loop emphasizes collaboration, context, and continuity in care. Rather than devoting in-person visits to basic data collection or medication reconciliation, clinicians engage with patients to interpret high-level trends derived from

[22] Hypertensive Crisis: A severe increase in blood pressure that can lead to stroke, heart attack, or other critical conditions.

[23] A model of operation in which human decision-makers are involved directly in the AI decision-making pathway.

continuous monitoring. These discussions focus on modifiable factors such as diet, physical activity, sleep quality, and stress management, enabling a more holistic approach to blood pressure control.

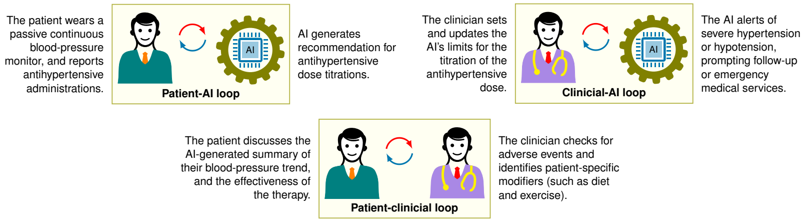

The patient wears a passive continuous blood-pressure monitor, and reports antihypertensive administrations.
AI
Patient-AI loop
AI generates recommendation for antihypertensive dose titrations.
The clinician sets and updates the AI's limits for the titration of the antihypertensive dose.
AI
Clinicial-AI loop
The AI alerts of severe hypertension or hypotension, prompting follow-up or emergency medical services.
The patient discusses the AI-generated summary of their blood-pressure trend, and the effectiveness of the therapy.
Patient-clinicial loop
The clinician checks for adverse events and identifies patient-specific modifiers (such as diet and exercise).

Figure 13.11: ClinAIOps interactive loop. Source: E. Chen et al. (2023).

The dynamic nature of continuous data allows for flexible scheduling of appointments based on clinical need rather than fixed intervals. For example, patients exhibiting stable blood pressure trends may be seen less frequently, while those experiencing variability may receive more immediate follow-up. This adaptive cadence enhances resource efficiency while preserving care quality.

By offloading routine monitoring and dose titration to AI-assisted systems, clinicians are better positioned to offer personalized counseling and targeted interventions. The result is a more meaningful patient-clinician relationship that supports shared decision-making and long-term wellness. This loop exemplifies how ClinAIOps frameworks can shift clinical interactions from transactional to transformational—supporting proactive care, patient empowerment, and improved health outcomes.

13.7.5.3 MLOps vs. ClinAIOps Comparison

The hypertension case study illustrates why traditional MLOps frameworks are often insufficient for high-stakes, real-world domains such as clinical healthcare. While conventional MLOps excels at managing the technical lifecycle of machine learning models, including training, deployment, and monitoring, it generally lacks the constructs necessary for coordinating human decision-making, managing clinical workflows, and safeguarding ethical accountability.

In contrast, the ClinAIOps framework extends beyond technical infrastructure to support complex sociotechnical systems[24]. Rather than treating the model as the final decision-maker, ClinAIOps embeds machine learning into a broader context where clinicians, patients, and systems stakeholders collaboratively shape treatment decisions.

Several limitations of a traditional MLOps approach become apparent when applied to a clinical setting like hypertension management:

- **Data availability and feedback**: Traditional pipelines rely on pre-collected datasets. ClinAIOps enables ongoing data acquisition and iterative feedback from clinicians and patients.
- **Trust and interpretability**: MLOps may lack transparency mechanisms for end users. ClinAIOps maintains clinician oversight, ensuring recommendations remain actionable and trustworthy.

[24] Sociotechnical System: An approach considering both social and technical aspects of organizational structures, prioritizing human well-being and system performance.

- **Behavioral and motivational factors**: MLOps focuses on model outputs. ClinAIOps recognizes the need for patient coaching, adherence support, and personalized engagement.
- **Safety and liability**: MLOps does not account for medical risk. ClinAIOps retains human accountability and provides structured boundaries for autonomous decisions.
- **Workflow integration**: Traditional systems may exist in silos. ClinAIOps aligns incentives and communication across stakeholders to ensure clinical adoption.

As shown in Table 13.6, the key distinction lies in how ClinAIOps integrates technical systems with human oversight, ethical principles, and care delivery processes. Rather than replacing clinicians, the framework augments their capabilities while preserving their central role in therapeutic decision-making.

Table 13.6: Comparison of MLOps versus AI operations for clinical use.

	Traditional MLOps	ClinAIOps
Focus	ML model development and deployment	Coordinating human and AI decision-making
Stakeholders	Data scientists, IT engineers	Patients, clinicians, AI developers
Feedback loops	Model retraining, monitoring	Patient-AI, clinician-AI, patient-clinician
Objective	Operationalize ML deployments	Optimize patient health outcomes
Processes	Automated pipelines and infrastructure	Integrates clinical workflows and oversight
Data considerations	Building training datasets	Privacy, ethics, protected health information
Model validation	Testing model performance metrics	Clinical evaluation of recommendations
Implementation	Focuses on technical integration	Aligns incentives of human stakeholders

Successfully deploying AI in complex domains such as healthcare requires more than developing and operationalizing performant machine learning models. As demonstrated by the hypertension case, effective integration depends on aligning AI systems with clinical workflows, human expertise, and patient needs. Technical performance alone is insufficient—deployment must account for ethical oversight, stakeholder coordination, and continuous adaptation to dynamic clinical contexts.

The ClinAIOps framework addresses these requirements by introducing structured, multi-stakeholder feedback loops that connect patients, clinicians, and AI developers. These loops enable human oversight, reinforce accountability, and ensure that AI systems adapt to evolving health data and patient responses. Rather than replacing human decision-makers, AI is positioned as an augmentation layer—enhancing the precision, personalization, and scalability of care.

By embedding AI within collaborative clinical ecosystems, frameworks like ClinAIOps create the foundation for trustworthy, responsive, and effective machine learning systems in high-stakes environments. This perspective reframes AI not as an isolated technical artifact, but as a component of a broader sociotechnical system designed to advance health outcomes and healthcare delivery.

13.8 Conclusion

The operationalization of machine learning is a complex, systems-oriented endeavor that extends far beyond training and deploying models. MLOps provides the methodological and infrastructural foundation for managing the full lifecycle of ML systems—from data collection and preprocessing to deployment, monitoring, and continuous refinement. By drawing on principles from software engineering, DevOps, and data science, MLOps offers the practices needed to achieve scalability, reliability, and resilience in real-world environments.

This chapter has examined the core components of MLOps, highlighting key challenges such as data quality, reproducibility, infrastructure automation, and organizational coordination. We have emphasized the importance of operational maturity, where model-centric development evolves into system-level engineering supported by robust processes, tooling, and feedback loops. Through detailed case studies in domains such as wearable computing and healthcare, we have seen how MLOps must adapt to specific operational contexts, technical constraints, and stakeholder ecosystems.

As we transition to subsequent chapters, we shift our focus toward emerging frontiers in operational practice, including on-device learning, privacy and security, responsible AI, and sustainable systems. Each of these domains introduces unique constraints that further shape how machine learning must be engineered and maintained in practice. These topics build on the foundation laid by MLOps, extending it into specialized operational regimes.

Ultimately, operational excellence in machine learning is not a fixed endpoint but a continuous journey. It requires cross-disciplinary collaboration, rigorous engineering, and a commitment to long-term impact. By approaching ML systems through the lens of MLOps, which are grounded in systems thinking and guided by ethical and societal considerations, we can build solutions that are not only technically sound but also trustworthy, maintainable, and meaningful in their real-world applications.

As the chapters ahead explore these evolving dimensions of machine learning systems, the central lesson remains clear: building models is only the beginning. The enduring challenge and opportunity lies in building systems that are adaptive, responsible, and effective in the face of complexity, uncertainty, and change.

13.9 Resources

ℹ Slides

- MLOps, DevOps, and AIOps.
- MLOps overview.
- Tiny MLOps.
- MLOps: a use case.
- MLOps: Key Activities and Lifecycle.
- ML Lifecycle.

- Scaling TinyML: Challenges and Opportunities.
- Training Operationalization:
  - Training Ops: CI/CD trigger.
  - Continuous Integration.
  - Continuous Deployment.
  - Production Deployment.
  - Production Deployment: Online Experimentation.
  - Training Ops Impact on MLOps.
- Model Deployment:
  - Scaling ML Into Production Deployment.
  - Containers for Scaling ML Deployment.
  - Challenges for Scaling TinyML Deployment: Part 1.
  - Challenges for Scaling TinyML Deployment: Part 2.
  - Model Deployment Impact on MLOps.

❗ Videos

- Video 5
- Video 6
- Video 7

🔥 Exercises

- Coming soon.

Chapter 14

On-Device Learning

Figure 14.1: *DALL·E 3 Prompt: Drawing of a smartphone with its internal components exposed, revealing diverse miniature engineers of different genders and skin tones actively working on the ML model. The engineers, including men, women, and non-binary individuals, are tuning parameters, repairing connections, and enhancing the network on the fly. Data flows into the ML model, being processed in real-time, and generating output inferences.*

Purpose

How does enabling learning directly on edge devices reshape machine learning system design, and what strategies support adaptation under resource constraints?

The shift toward on-device learning marks a significant evolution in the deployment and maintenance of machine learning systems. Rather than relying exclusively on centralized infrastructure, models are now increasingly expected to adapt in situ—updating and improving directly on the devices where they operate. This approach introduces a new design space, where training must occur within stringent constraints on memory, compute, energy, and data availability. In these settings, the balance between model adaptability, system efficiency, and deployment scalability becomes critical. This chapter examines the architectural, algorithmic, and infrastructure-level techniques that enable effective learning on the edge, and outlines the principles required to support autonomous model improvement in resource-constrained environments.

Learning Objectives

- Understand on-device learning and how it differs from cloud-based training
- Recognize the benefits and limitations of on-device learning
- Examine strategies to adapt models through complexity reduction, optimization, and data compression
- Understand related concepts like federated learning and transfer learning
- Analyze the security implications of on-device learning and mitigation strategies

14.1 Overview

Machine learning systems have traditionally treated model training and model inference as distinct phases, often separated by both time and infrastructure. Training occurs in the cloud, leveraging large-scale compute clusters and curated datasets, while inference is performed downstream on deployed models—typically on user devices or edge servers. However, this separation is beginning to erode. Increasingly, devices are being equipped not just to run inference, but to adapt, personalize, and improve models locally.

On-device learning refers to the process of training or adapting machine learning models directly on the device where they are deployed. This capability opens the door to systems that can personalize models in response to user behavior, operate without cloud connectivity, and respect stringent privacy constraints by keeping data local. It also introduces a new set of challenges: devices have limited memory, computational power, and energy. Furthermore, training data is often sparse, noisy, or non-independent across users. These limitations necessitate a fundamental rethinking of training algorithms, system architecture, and deployment strategies.

Definition of On-Device Learning

On-Device Learning is the *local adaptation or training* of machine learning models directly on deployed hardware devices, without reliance on continuous connectivity to centralized servers. It enables *personalization, privacy preservation, and autonomous operation* by leveraging user-specific data collected in situ. On-device learning systems must operate under *tight constraints on compute, memory, energy, and data availability*, requiring specialized methods for model optimization, training efficiency, and data representation. As on-device learning matures, it increasingly in-

> corporates *federated collaboration, lifelong adaptation, and secure execution*, expanding the frontier of intelligent edge computing.

This chapter explores the principles and systems design considerations underpinning on-device learning. It begins by examining the motivating applications that necessitate learning on the device, followed by a discussion of the unique hardware constraints introduced by embedded and mobile environments. The chapter then develops a taxonomy of strategies for adapting models, algorithms, and data pipelines to these constraints. Particular emphasis is placed on distributed and collaborative methods, such as federated learning, which enable decentralized training without direct data sharing. The chapter concludes with an analysis of outstanding challenges, including issues related to reliability, system validation, and the heterogeneity of deployment environments.

14.2 Deployment Drivers

Machine learning systems have traditionally relied on centralized training pipelines, where models are developed and refined using large, curated datasets and powerful cloud-based infrastructure (Jeffrey Dean and Ghemawat 2008). Once trained, these models are deployed to client devices for inference. While this separation has served most use cases well, it imposes limitations in settings where local data is dynamic, private, or personalized. On-device learning challenges this model by enabling systems to train or adapt directly on the device, without relying on constant connectivity to the cloud.

14.2.1 On-Device Learning Benefits

Traditional machine learning systems rely on a clear division of labor between model training and inference. Training is performed in centralized environments with access to high-performance compute resources and large-scale datasets. Once trained, models are distributed to client devices, where they operate in a static inference-only mode. While this centralized paradigm has been effective in many deployments, it introduces limitations in settings where data is user-specific, behavior is dynamic, or connectivity is intermittent.

On-device learning refers to the capability of a deployed device to perform model adaptation using locally available data. This shift from centralized to decentralized learning is motivated by four key considerations: personalization, latency and availability, privacy, and infrastructure efficiency (T. Li et al. 2020).

Personalization is a primary motivation. Deployed models often encounter usage patterns and data distributions that differ substantially from their training environments. Local adaptation enables models to refine behavior in response to user-specific data—capturing linguistic preferences, physiological baselines, sensor characteristics, or environmental conditions. This is particularly important in applications with high inter-user variability, where a single global model may fail to serve all users equally well.

Latency and availability further justify local learning. In edge computing scenarios, connectivity to centralized infrastructure may be unreliable, delayed, or intentionally limited to preserve bandwidth or reduce energy usage. On-device learning enables autonomous improvement of models even in fully offline or delay-sensitive contexts, where round-trip updates to the cloud are infeasible.

Privacy is another critical factor. Many applications involve sensitive or regulated data, including biometric measurements, typed input, location traces, or health information. Transmitting such data to the cloud introduces privacy risks and compliance burdens. Local learning mitigates these concerns by keeping raw data on the device and operating within privacy-preserving boundaries—potentially aiding adherence to regulations such as GDPR[0], HIPAA[1], or region-specific data sovereignty laws.

Infrastructure efficiency also plays a role. Centralized training pipelines require substantial backend infrastructure to collect, store, and process user data. At scale, this introduces bottlenecks in bandwidth, compute capacity, and energy consumption. By shifting learning to the edge, systems can reduce communication costs and distribute training workloads across the deployment fleet, relieving pressure on centralized resources.

These motivations are grounded in the broader concept of knowledge transfer, where a pretrained model transfers useful representations to a new task or domain. As depicted in Figure 14.2, knowledge transfer can occur between closely related tasks (e.g., playing different board games or musical instruments), or across domains that share structure (e.g., from riding a bicycle to driving a scooter). In the context of on-device learning, this means leveraging a model pretrained in the cloud and adapting it efficiently to a new context using only local data and limited updates. The figure highlights the key idea: pretrained knowledge enables fast adaptation without relearning from scratch, even when the new task diverges in input modality or goal.

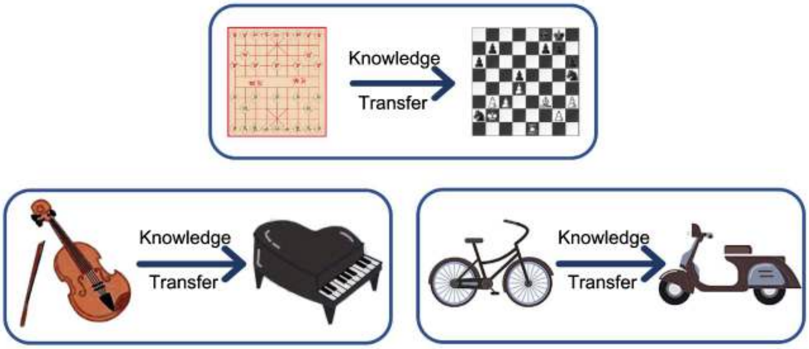


Figure 14.2: Conceptual illustration of knowledge transfer across tasks and domains. The left side shows a pretrained model adapting to a new task, while the right side illustrates transfer across different domains.

This conceptual shift, which is enabled by transfer learning and adaptation, is essential for real-world on-device applications. Whether adapting a language model for personal typing preferences, adjusting gesture recognition to an individual's movement patterns, or recalibrating a sensor model in a changing environment, on-device learning allows systems to remain responsive, efficient, and user-aligned over time.

[0] GDPR: General Data Protection Regulation, a legal framework that sets guidelines for the collection and processing of personal information in the EU.

[1] HIPAA: Health Insurance Portability and Accountability Act, U.S. legislation that provides data privacy and security provisions for safeguarding medical information.

14.2.2 Application Domains

The motivations for on-device learning are most clearly illustrated by examining the application domains where its benefits are both tangible and necessary. These domains span consumer technologies, healthcare, industrial systems, and embedded applications, each presenting scenarios where local adaptation is preferable, or even required, for effective machine learning deployment.

Mobile input prediction is a mature example of on-device learning in action. In systems such as smartphone keyboards, predictive text and autocorrect features benefit substantially from continuous local adaptation. User typing patterns are highly personalized and evolve dynamically, making centralized static models insufficient. On-device learning enables language models to finetune their predictions directly on the device, without transmitting keystroke data to external servers. This approach not only supports personalization but also aligns with privacy-preserving design principles.

For instance, Google's Gboard employs federated learning to improve shared models across a large population of users while keeping raw data local to each device (Hard et al. 2018). As shown in Figure 14.3, different prediction strategies illustrate how local adaptation can operate in real-time: next-word prediction (NWP) suggests likely continuations based on prior text, while Smart Compose leverages on-the-fly rescoring to offer dynamic completions, showcasing the sophistication of local inference mechanisms.

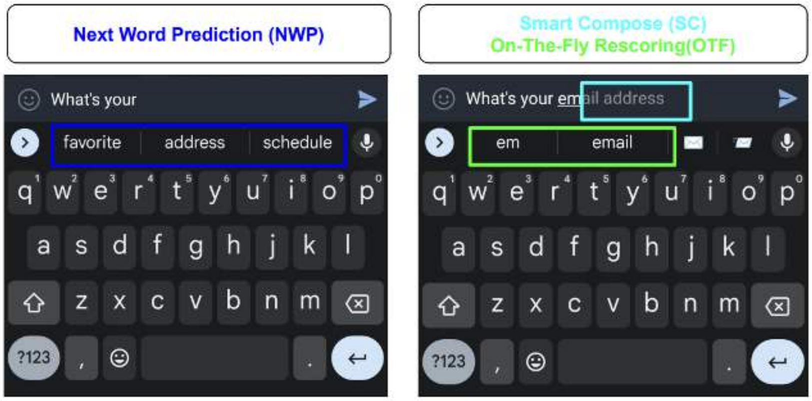

Next Word Prediction (NWP)
Smart Compose (SC)
On-The-Fly Rescoring(OTF)
What's your
favorite
address
schedule
What's your email address
em
email

Figure 14.3: Illustration of two input prediction modes in Gboard. Left: Next Word Prediction (NWP). Right: Smart Compose (SC) with On-The-Fly Rescoring (OTF).

Wearable and health monitoring devices also present strong use cases. These systems often rely on real-time data from accelerometers, heart rate sensors, or electrodermal activity monitors. However, physiological baselines vary significantly between individuals. On-device learning allows models to adapt to these baselines over time, improving the accuracy of activity recognition, stress detection, and sleep staging. Moreover, in regulated healthcare environments, patient data must remain localized due to privacy laws, further reinforcing the need for edge-local adaptation.

Wake-word detection and voice interfaces illustrate another critical scenario. Devices such as smart speakers and earbuds must recognize voice commands quickly and accurately, even in noisy or dynamic acoustic environments. Local training enables models to adapt to the user's voice profile and ambient context, reducing false positives and missed detections. This kind of adaptation is particularly valuable in far-field audio settings, where microphone configurations and room acoustics vary widely across deployments.

Industrial IoT[2] and remote monitoring systems also benefit from local learning capabilities. In applications such as agricultural sensing, pipeline monitoring, or environmental surveillance, connectivity to centralized infrastructure may be limited or costly. On-device learning allows these systems to detect anomalies, adjust thresholds, or adapt to seasonal trends without continuous communication with the cloud. This capability is critical for maintaining autonomy and reliability in edge-deployed sensor networks.

Embedded computer vision systems, including those in robotics, AR/VR, and smart cameras, present additional opportunities. These systems often operate in novel or evolving environments that differ significantly from training conditions. On-device adaptation allows models to recalibrate to new lighting conditions, object appearances, or motion patterns, maintaining task accuracy over time.

Each of these domains highlights a common pattern: the deployment environment introduces variation or uncertainty that cannot be fully anticipated during centralized training. On-device learning offers a mechanism for adapting models in place, enabling systems to improve continuously in response to local conditions. These examples also reveal a critical design requirement: learning must be performed efficiently, privately, and reliably under significant resource constraints. The following section formalizes these constraints and outlines the system-level considerations that shape the design of on-device learning solutions.

14.2.3 Training Paradigms

Most machine learning systems today follow a centralized learning paradigm. Models are trained in data centers using large-scale, curated datasets aggregated from many sources. Once trained, these models are deployed to client devices in a static form, where they perform inference without further modification. Updates to model parameters, either to incorporate new data or to improve generalization, are handled periodically through offline retraining, often using newly collected or labeled data sent back from the field.

This centralized model of learning offers numerous advantages: high-performance computing infrastructure, access to diverse data distributions, and robust debugging and validation pipelines. However, it also depends on reliable data transfer, trust in data custodianship, and infrastructure capable of managing global updates across a fleet of devices. As machine learning is deployed into increasingly diverse and distributed environments, the limitations of this approach become more apparent.

In contrast, on-device learning is inherently decentralized. Each device maintains its own copy of a model and adapts it locally using data that is typically unavailable to centralized infrastructure. Training occurs on-device, often asynchronously and under varying resource conditions. Data never leaves the

[2] Industrial Internet of Things (IoT): Network of physical objects, including devices, vehicles, and buildings, that use sensors and software to collect and exchange data.

device, reducing exposure but also complicating coordination. Devices may differ substantially in their hardware capabilities, runtime environments, and patterns of use, making the learning process heterogeneous and difficult to standardize.

This decentralized nature introduces unique systems challenges. Devices may operate with different versions of the model, leading to inconsistencies in behavior. Evaluation and validation become more complex, as there is no central point from which to measure performance (H. B. McMahan et al. 2017). Model updates must be carefully managed to prevent degradation, and safety guarantees become harder to enforce in the absence of centralized testing.

At the same time, decentralization introduces opportunities. It allows for personalization without centralized oversight, supports learning in disconnected or bandwidth-limited environments, and reduces the cost of infrastructure for model updates. It also raises important questions of how to coordinate learning across devices, whether through periodic synchronization, federated aggregation, or hybrid approaches that combine local and global objectives.

The move from centralized to decentralized learning represents more than a shift in deployment architecture—it fundamentally reshapes the design space for machine learning systems. In centralized training, data is aggregated from many sources and processed in large-scale data centers, where models are trained, validated, and then deployed in a static form to edge devices. In contrast, on-device learning introduces a decentralized paradigm: models are updated directly on client devices using local data, often asynchronously and under diverse hardware conditions. This change reduces reliance on cloud infrastructure and enhances personalization and privacy, but it also introduces new coordination and validation challenges.

On-device learning emerges as a response to the limitations of centralized machine learning workflows. As illustrated in Figure 14.4, the traditional paradigm (A) involves training a model on aggregated cloud-based data before pushing it to client devices for static inference. This architecture works well when centralized data collection is feasible, network connectivity is reliable, and model generalization across users is sufficient. However, it falls short in scenarios where data is highly personalized, privacy-sensitive, or collected in environments with limited connectivity.

In contrast, once the model is deployed, local differences begin to emerge. Region B depicts the process by which each device collects its own data stream, which is often non-IID[3] and noisy, and adapts the model to better reflect its specific operating context. This marks the shift from global generalization to local specialization, highlighting the autonomy and variability introduced by decentralized learning.

Figure 14.4 illustrates this shift. In region A, centralized learning begins with cloud-based training on aggregated data, followed by deployment to client devices. Region B marks the transition to local learning: devices begin collecting data, which is frequently non-IID, noisy, and unlabeled, and adapting their models based on individual usage patterns. Finally, region C depicts federated learning, in which client updates are periodically synchronized via aggregated model updates rather than raw data transfer, enabling privacy-preserving global refinement.

[3] Non-IID Data: Datasets where samples are not independently and identically distributed, often seen in personalized data streams.

This shift from centralized training to decentralized, adaptive learning reshapes how ML systems are designed and deployed. It enables learning in settings where connectivity is intermittent, data is user-specific, and personalization is essential—while introducing new challenges in update coordination, evaluation, and system robustness.

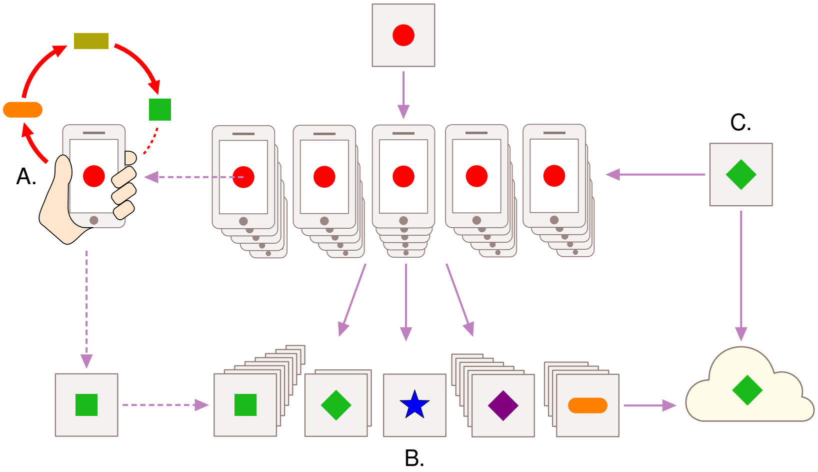


Figure 14.4: Comparison of centralized training versus decentralized, on-device learning workflows. Local updates are generated and applied at the edge before optional synchronization with a global model.

14.3 Design Constraints

Enabling learning on the device requires rethinking conventional assumptions about where and how machine learning systems operate. In centralized environments, models are trained with access to extensive compute infrastructure, large and curated datasets, and generous memory and energy budgets. At the edge, none of these assumptions hold. Instead, on-device learning must navigate a constrained design space shaped by the structure of the model, the nature of the available data, and the computational capabilities of the deployment platform.

These three dimensions, the model, the data, and the computational resources, form the foundation of any on-device learning system. Each imposes distinct limitations that influence algorithmic design and system architecture. The model must be compact enough to fit within memory and storage bounds, yet expressive enough to support adaptation. The data is local, often sparse, unlabeled, and non-IID, requiring robust and efficient learning procedures. The compute environment is resource-constrained, often lacking support for floating-point operations or backpropagation primitives. These constraints are not merely technical—they reflect the realities of deploying machine learning systems in the wild. Devices may be battery-powered, have limited connectivity, and operate in unpredictable environments. They may also be heterogeneous, with different hardware capabilities and software stacks. As a result, on-device learning must be designed to accommodate these variations while still delivering reliable performance.

Figure 14.5 illustrates a pipeline that combines offline pre-training with online adaptive learning on resource-constrained IoT devices. The system first undergoes meta-training with generic data. During deployment, device-specific constraints such as data availability, compute, and memory shape the adaptation strategy by ranking and selecting layers and channels to update. This enables efficient on-device learning within limited resource envelopes.

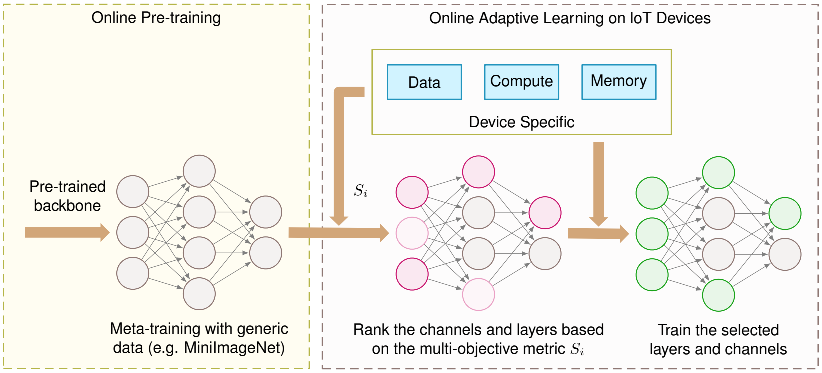

Online Pre-training
Pre-trained backbone
Meta-training with generic data (e.g. MiniImageNet)
Online Adaptive Learning on IoT Devices
Data
Compute
Memory
Device Specific
S_i
Rank the channels and layers based on the multi-objective metric S_i
Train the selected layers and channels

Figure 14.5: On-device adaptation framework.

14.3.1 Model Constraints

The structure and size of the machine learning model directly influence the feasibility of on-device training. Unlike cloud-deployed models that can span billions of parameters and rely on multi-gigabyte memory budgets, models intended for on-device learning must conform to tight constraints on memory, storage, and computational complexity. These constraints apply not only at inference time, but also during training, where additional resources are needed for gradient computation, parameter updates, and optimizer state.

For example, the MobileNetV2 architecture, commonly used in mobile vision tasks, requires approximately 14 MB of storage in its standard configuration. While this is feasible for modern smartphones, it far exceeds the memory available on embedded microcontrollers such as the Arduino Nano 33 BLE Sense, which provides only 256 KB of SRAM and 1 MB of flash storage. In such platforms, even a single layer of a typical convolutional neural network may exceed available RAM during training due to the need to store intermediate feature maps.

In addition to storage constraints, the training process itself expands the effective memory footprint. Standard backpropagation requires caching activations for each layer during the forward pass, which are then reused during gradient computation in the backward pass. For a 10-layer convolutional model processing 64×64 images, the required memory may exceed 1–2 MB—well beyond the SRAM capacity of most embedded systems.

Model complexity also affects runtime energy consumption and thermal limits. In systems such as smartwatches or battery-powered wearables, sustained model training can deplete energy reserves or trigger thermal throttling[4]. Training a full model using floating-point operations on these devices is often

[4] Reduction in computing performance to prevent overheating in electronic devices.

infeasible. This limitation has motivated the development of ultra-lightweight model variants, such as MLPerf Tiny[5] benchmark networks (C. Banbury et al. 2021), which fit within 100–200 KB and can be adapted using only partial gradient updates.

The model architecture itself must also be designed with on-device learning in mind. Many conventional architectures, such as transformers or large convolutional networks, are not well-suited for on-device adaptation due to their size and complexity. Instead, lightweight architectures such as MobileNets, SqueezeNet, and EfficientNet have been developed specifically for resource-constrained environments. These models use techniques such as depthwise separable convolutions, bottleneck layers, and quantization to reduce memory and compute requirements while maintaining performance.

These architectures are often designed to be modular, allowing for easy adaptation and fine-tuning. For example, MobileNets (A. G. Howard et al. 2017b) can be configured with different width multipliers and resolution settings to balance performance and resource usage. This flexibility is critical for on-device learning, where the model must adapt to the specific constraints of the deployment environment.

14.3.2 Data Constraints

The nature of data available to on-device learning systems differs significantly from the large, curated, and centrally managed datasets typically used in cloud-based training. At the edge, data is locally collected, temporally sparse, and often unstructured or unlabeled. These characteristics introduce challenges in volume, quality, and statistical distribution, all of which affect the reliability and generalizability of learning on the device.

Data volume is typically limited due to storage constraints and the nature of user interaction. For example, a smart fitness tracker may collect motion data only during physical activity, generating relatively few labeled samples per day. If a user wears the device for just 30 minutes of exercise, only a few hundred data points might be available for training, compared to the thousands typically required for supervised learning in controlled environments.

Moreover, on-device data is frequently non-IID (non-independent and identically distributed) (Y. Zhao et al. 2018). Consider a voice assistant deployed in different households: one user may issue commands in English with a strong regional accent, while another might speak a different language entirely. The local data distribution is highly user-specific and may differ substantially from the training distribution of the initial model. This heterogeneity complicates both model convergence and the design of update mechanisms that generalize well across devices.

Label scarcity presents an additional obstacle. Most edge-collected data is unlabeled by default. In a smartphone camera, for instance, the device may capture thousands of images, but only a few are associated with user actions (e.g., tagging or favoriting), which could serve as implicit labels. In many applications, including detecting anomalies in sensor data and adapting gesture recognition models, labels may be entirely unavailable, making traditional supervised learning infeasible without additional methods.

[5] MLPerf Tiny: A benchmark suite for evaluating the performance of ultra-low power machine learning systems in real-world scenarios.

Noise and variability further degrade data quality. Embedded systems such as environmental sensors or automotive ECUs[6] may experience fluctuations in sensor calibration, environmental interference, or mechanical wear, leading to corrupted or drifting input signals over time. Without centralized validation, these errors may silently degrade learning performance if not detected and filtered appropriately.

Finally, data privacy and security concerns are paramount in many on-device learning applications. Sensitive information, such as health data or user interactions, must be protected from unauthorized access. This requirement often precludes the use of traditional data-sharing methods, such as uploading raw data to a central server for training. Instead, on-device learning must rely on techniques that allow for local adaptation without exposing sensitive information.

14.3.3 Compute Constraints

On-device learning must operate within the computational envelope of the target hardware platform, which ranges from low-power embedded microcontrollers to mobile-class processors found in smartphones and wearables. These systems differ substantially from the large-scale GPU or TPU infrastructure used in cloud-based training. They impose strict limits on instruction throughput, parallelism, and architectural support for training-specific operations, all of which shape the design of feasible learning strategies.

On the embedded end of the spectrum, devices such as the STM32F4 or ESP32 microcontrollers offer only a few hundred kilobytes of SRAM and lack hardware support for floating-point operations (P. W. D. S. Lai 2020). These constraints preclude the use of conventional deep learning libraries and require models to be carefully designed for integer arithmetic and minimal runtime memory allocation. In such cases, even small models require tailored techniques, including quantization-aware training and selective parameter updates, to execute training loops without exceeding memory or power budgets. For example, the STM32F4 microcontroller can run a simple linear regression model with a few hundred parameters, but training even a small convolutional neural network would exceed its memory capacity. In these environments, training is often limited to simple algorithms such as stochastic gradient descent (SGD) or k-means clustering, which can be implemented using integer arithmetic and minimal memory overhead.

In contrast, mobile-class hardware, including the Qualcomm Snapdragon, Apple Neural Engine, and Google Tensor SoC, provides significantly more compute power, often with dedicated AI accelerators and optimized support for 8-bit or mixed-precision matrix operations. These platforms can support more complex training routines, including full backpropagation over compact models, though they still fall short of the computational throughput and memory bandwidth available in centralized data centers. For instance, training a lightweight transformer on a smartphone is feasible but must be tightly bounded in both time and energy consumption to avoid degrading the user experience.

Compute constraints are especially salient in real-time or battery-operated systems. In a smartphone-based speech recognizer, on-device adaptation must

[6] Electronic Control Unit (ECU): A device that controls one or more of the electrical systems or subsystems in a vehicle.

not interfere with inference latency or system responsiveness. Similarly, in wearable medical monitors, training must occur opportunistically, during periods of low activity or charging, to preserve battery life and avoid thermal issues.

14.4 Model Adaptation

Adapting a machine learning model on the device requires revisiting a core assumption of conventional training: that the entire model must be updated. In resource-constrained environments, this assumption becomes infeasible due to memory, compute, and energy limitations. Instead, modern approaches to on-device learning often focus on minimizing the scope of adaptation, updating only a subset of model parameters while reusing the majority of the pretrained architecture. These approaches leverage the power of transfer learning, starting with a model pretrained (usually offline on large datasets) and efficiently specializing it using the limited local data and compute resources available at the edge. This strategy is particularly effective when the pretrained model has already learned useful representations that can be adapted to new tasks or domains. By freezing most of the model parameters and only updating a small subset, we can achieve significant reductions in memory and compute requirements while still allowing for meaningful adaptation.

This strategy reduces both computational overhead and memory usage during training, enabling efficient local updates on devices ranging from smartphones to embedded microcontrollers. The central idea is to retain most of the model as a frozen backbone, while introducing lightweight, adaptable components, including bias-only updates, residual adapters, or task-specific layers, that can capture local variations in data. These techniques enable personalized or environment-aware learning without incurring the full cost of end-to-end finetuning.

In the sections that follow, we examine how minimal adaptation strategies are designed, the tradeoffs they introduce, and their role in enabling practical on-device learning.

14.4.1 Weight Freezing

One of the simplest and most effective strategies for reducing the cost of on-device learning is to freeze the majority of a model's parameters and adapt only a minimal subset. A widely used approach is bias-only adaptation, in which all weights are fixed and only the bias terms, which are typically scalar offsets applied after linear or convolutional layers, are updated during training. This significantly reduces the number of trainable parameters, simplifies memory management during backpropagation, and helps mitigate overfitting when data is sparse or noisy.

Consider a standard neural network layer:

$$y = Wx + b$$

where $W \in \mathbb{R}^{m \times n}$ is the weight matrix, $b \in \mathbb{R}^m$ is the bias vector, and $x \in \mathbb{R}^n$ is the input. In full training, gradients are computed for both W and b. In bias-only adaptation, we constrain:

$$\frac{\partial \mathcal{L}}{\partial W} = 0, \quad \frac{\partial \mathcal{L}}{\partial b} \neq 0$$

so that only the bias is updated via gradient descent:

$$b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}$$

This drastically reduces the number of stored gradients and optimizer states, enabling training to proceed even under memory-constrained conditions. On embedded devices that lack floating-point units, this reduction can be critical to enabling on-device learning at all.

The code snippet in Listing 14.1 demonstrates how to implement bias-only adaptation in PyTorch.

Listing 14.1: Bias-only adaptation in PyTorch

```
# Freeze all parameters
for name, param in model.named_parameters():
    param.requires_grad = False

# Enable gradients for bias parameters only
for name, param in model.named_parameters():
    if 'bias' in name:
        param.requires_grad = True
```

This pattern ensures that only bias terms participate in the backward pass and optimizer update. It is particularly useful when adapting pretrained models to user-specific or device-local data.

This technique underpins TinyTL, a framework explicitly designed to enable efficient adaptation of deep neural networks on microcontrollers and other memory-limited platforms. Rather than updating all network parameters during training, TinyTL freezes both the convolutional weights and the batch normalization statistics, training only the bias terms and, in some cases, lightweight residual components. This architectural shift drastically reduces memory usage during backpropagation, since the largest tensors, which are intermediate activations, no longer need to be stored for gradient computation.

Figure 14.6 illustrates the architectural differences between a standard model and the TinyTL approach. In the conventional baseline architecture, all layers are trainable, and backpropagation requires storing intermediate activations for the full network. This significantly increases the memory footprint, which quickly becomes infeasible on edge devices with only a few hundred kilobytes of SRAM.

In contrast, the TinyTL architecture freezes all weights and updates only the bias terms inserted after convolutional layers. These bias modules are lightweight and require minimal memory, enabling efficient training with a

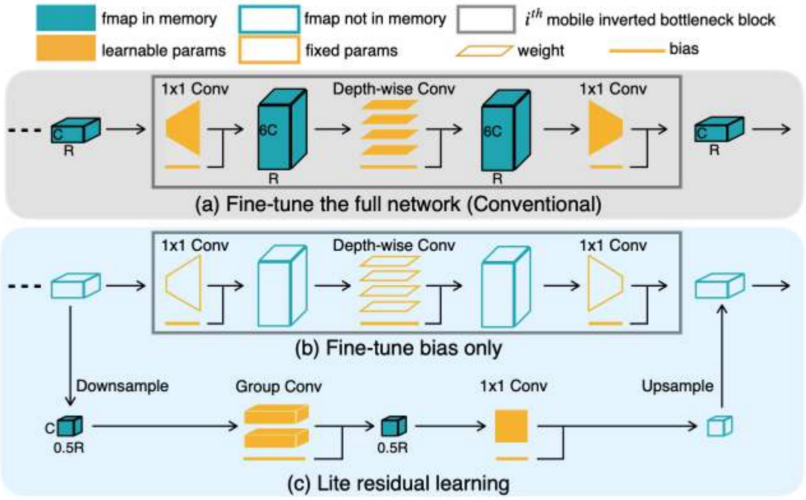


Figure 14.6: TinyTL freezes weights and batch norm statistics, adapting only the biases and lightweight components to enable memory-efficient on-device training.

drastically reduced memory footprint. The frozen convolutional layers act as a fixed feature extractor, and only the trainable bias components are involved in adaptation. By avoiding storage of full activation maps and limiting the number of updated parameters, TinyTL enables on-device training under severe resource constraints.

Because the base model remains unchanged, TinyTL assumes that the pretrained features are sufficiently expressive for downstream tasks. The bias terms allow for minor but meaningful shifts in model behavior, particularly for personalization tasks. When domain shift is more significant, TinyTL can optionally incorporate small residual adapters to improve expressivity, all while preserving the system's tight memory and energy profile.

These design choices allow TinyTL to reduce training memory usage by more than 10×. For instance, adapting a MobileNetV2 model using TinyTL can reduce the number of updated parameters from over 3 million to fewer than 50,000. Combined with quantization, this enables local adaptation on devices with only a few hundred kilobytes of memory—making on-device learning truly feasible in constrained environments.

14.4.2 Residual and Low-Rank Updates

Bias-only updates offer a lightweight path for on-device learning, but they are limited in representational flexibility. When the frozen model does not align well with the target distribution, it may be necessary to allow more expressive adaptation—without incurring the full cost of weight updates. One solution is to introduce residual adaptation modules (Houlsby et al. 2019),[7] or low-rank parameterizations[8], which provide a middle ground between static backbones[9] and full fine-tuning (E. J. Hu et al. 2021).

These methods extend a frozen model by adding trainable layers, which are typically small and computationally inexpensive, that allow the network to respond to new data. The main body of the network remains fixed, while only the

[7] Residual Adaptation Modules: Layers added to existing networks to improve adaptability without extensive retraining.

[8] Low-rank Parameterizations: Techniques that decompose parameters into low-rank matrices to save computation.

[9] Static Backbones: Unchangeable core parts of a neural network model, typically pre-trained.

added components are optimized. This modularity makes the approach well-suited for on-device adaptation in constrained settings, where small updates must deliver meaningful changes.

14.4.2.1 Adapter-Based Adaptation

A common implementation involves inserting adapters, which are small residual bottleneck layers, between existing layers in a pretrained model. Consider a hidden representation h passed between layers. A residual adapter introduces a transformation:

$$h' = h + A(h)$$

where $A(\cdot)$ is a trainable function, typically composed of two linear layers with a nonlinearity:

$$A(h) = W_2\sigma(W_1 h)$$

with $W_1 \in \mathbb{R}^{r\times d}$ and $W_2 \in \mathbb{R}^{d\times r}$, where $r \ll d$. This bottleneck design ensures that only a small number of parameters are introduced per layer.

The adapters act as learnable perturbations on top of a frozen backbone. Because they are small and sparsely applied, they add negligible memory overhead, yet they allow the model to shift its predictions in response to new inputs.

14.4.2.2 Low-Rank Techniques

Another efficient strategy is to constrain weight updates themselves to a low-rank structure. Rather than updating a full matrix W, we approximate the update as:

$$\Delta W \approx UV^\top$$

where $U \in \mathbb{R}^{m\times r}$ and $V \in \mathbb{R}^{n\times r}$, with $r \ll \min(m,n)$. This reduces the number of trainable parameters from mn to $r(m+n)$. During adaptation, the new weight is computed as:

$$W_{\text{adapted}} = W_{\text{frozen}} + UV^\top$$

This formulation is commonly used in LoRA (Low-Rank Adaptation) techniques, originally developed for transformer models (E. J. Hu et al. 2021) but broadly applicable across architectures. Low-rank updates can be implemented efficiently on edge devices, particularly when U and V are small and fixed-point representations are supported (Listing 14.2).

This adapter adds a small residual transformation to a frozen layer. When inserted into a larger model, only the adapter parameters are trained.

Listing 14.2: Low-Rank Adapter Module

```python
class Adapter(nn.Module):
    def __init__(self, dim, bottleneck_dim):
        super().__init__()
        self.down = nn.Linear(dim, bottleneck_dim)
        self.up = nn.Linear(bottleneck_dim, dim)
        self.activation = nn.ReLU()

    def forward(self, x):
        return x + self.up(self.activation(self.down(x)))
```

14.4.2.3 Edge Personalization

Adapters are especially useful when a global model is deployed to many devices and must adapt to device-specific input distributions. For instance, in smartphone camera pipelines, environmental lighting, user preferences, or lens distortion may vary between users (Rebuffi, Bilen, and Vedaldi 2017). A shared model can be frozen and fine-tuned per-device using a few residual modules, allowing lightweight personalization without risking catastrophic forgetting[10]. In voice-based systems, adapter modules have been shown to reduce word error rates in personalized speech recognition without retraining the full acoustic model. They also allow easy rollback or switching between user-specific versions.

14.4.2.4 Tradeoffs

Residual and low-rank updates strike a balance between expressivity and efficiency. Compared to bias-only learning, they can model more substantial deviations from the pretrained task. However, they require more memory and compute—both for training and inference.

When considering residual and low-rank updates for on-device learning, several important tradeoffs emerge. First, these methods consistently demonstrate superior adaptation quality compared to bias-only approaches, particularly when deployed in scenarios involving significant distribution shifts[11] from the original training data (Quiñonero-Candela et al. 2008). This improved adaptability stems from their increased parameter capacity and ability to learn more complex transformations.

However, this enhanced adaptability comes at a cost. The introduction of additional layers or parameters inevitably increases both memory requirements and computational latency during forward and backward passes. While these increases are modest compared to full model training, they must be carefully considered when deploying to resource-constrained devices.

Additionally, implementing these adaptation techniques requires system-level support for dynamic computation graphs[12] and the ability to selectively inject trainable parameters. Not all deployment environments or inference engines may support such capabilities out of the box.

[10] Catastrophic Forgetting: A phenomenon where a neural network forgets previously learned information upon learning new data.

[11] Distribution shifts refer to changes in the input data's characteristics, which can affect model performance when different from the training data.

[12] Dynamic Computation Graphs: Structures that allow changes during runtime, enabling models to adapt structures based on input data.

Despite these considerations, residual adaptation techniques have proven particularly valuable in mobile and edge computing scenarios where devices have sufficient computational resources. For instance, modern smartphones and tablets can readily accommodate these adaptations while maintaining acceptable performance characteristics. This makes residual adaptation a practical choice for applications requiring personalization without the overhead of full model retraining.

14.4.3 Sparse Updates

Even when adaptation is restricted to a small number of parameters, including biases or adapter modules, training remains resource-intensive on constrained devices. One promising approach is to selectively update only a task-relevant subset of model parameters, rather than modifying the entire network or introducing new modules. This approach is known as task-adaptive sparse updating (X. Zhang, Song, and Tao 2020).

The key insight is that not all layers of a deep model contribute equally to performance gains on a new task or dataset. If we can identify a *minimal subset of parameters* that are most impactful for adaptation, we can train only those, reducing memory and compute costs while still achieving meaningful personalization.

14.4.3.1 Sparse Update Design

Let a neural network be defined by parameters $\theta = \{\theta_1, \theta_2, \ldots, \theta_L\}$ across L layers. In standard fine-tuning, we compute gradients and perform updates on all parameters:

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i}, \quad \text{for } i = 1, \ldots, L$$

In task-adaptive sparse updates, we select a small subset $\mathcal{S} \subset \{1, \ldots, L\}$ such that only parameters in $\mathcal{S}$ are updated:

$$\theta_i \leftarrow \begin{cases} \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i}, & \text{if } i \in \mathcal{S} \\ \theta_i, & \text{otherwise} \end{cases}$$

The challenge lies in selecting the optimal subset $\mathcal{S}$ given memory and compute constraints.

14.4.3.2 Layer Selection

A principled strategy for selecting $\mathcal{S}$ is to use contribution analysis—an empirical method that estimates how much each layer contributes to downstream performance improvement. For example, one can measure the marginal gain from updating each layer independently:

1. Freeze the entire model.
2. Unfreeze one candidate layer.
3. Finetune briefly and evaluate improvement in validation accuracy.

4. Rank layers by performance gain per unit cost (e.g., per KB of trainable memory).

This layer-wise profiling yields a ranking from which $\mathcal{S}$ can be constructed subject to a memory budget.

A concrete example is TinyTrain, a method designed to enable rapid adaptation on-device (C. Deng, Zhang, and Wu 2022). TinyTrain pretrains a model along with meta-gradients that capture which layers are most sensitive to new tasks. At runtime, the system dynamically selects layers to update based on task characteristics and available resources.

14.4.3.3 Code Fragment: Selective Layer Updating (PyTorch)

Listing 14.3: Selective layer updating in PyTorch

```python
## Assume model has named layers: ['conv1', 'conv2', 'fc']
## We selectively update only conv2 and fc

for name, param in model.named_parameters():
    if 'conv2' in name or 'fc' in name:
        param.requires_grad = True
    else:
        param.requires_grad = False
```

This pattern can be extended with profiling logic to select layers based on contribution scores or hardware profiles, as shown in Listing 14.3.

14.4.3.4 TinyTrain Personalization

Consider a scenario where a user wears an augmented reality headset that performs real-time object recognition. As lighting and environments shift, the system must adapt to maintain accuracy—but training must occur during brief idle periods or while charging.

TinyTrain enables this by using meta-training during offline preparation: the model learns not only to perform the task, but also which parameters are most important to adapt. Then, at deployment, the device performs task-adaptive sparse updates, modifying only a few layers that are most relevant for its current environment. This keeps adaptation fast, energy-efficient, and memory-aware.

14.4.3.5 Tradeoffs

Task-adaptive sparse updates introduce several important system-level considerations that must be carefully balanced. First, the overhead of contribution analysis, although primarily incurred during pretraining or initial profiling, represents a non-trivial computational cost. This overhead is typically acceptable since it occurs offline, but it must be factored into the overall system design and deployment pipeline.

Second, the stability of the adaptation process becomes critical when working with sparse updates. If too few parameters are selected for updating, the model may underfit the target distribution, failing to capture important local variations. This suggests the need for careful validation of the selected parameter subset before deployment, potentially incorporating minimum thresholds for adaptation capacity.

Third, the selection of updateable parameters must account for hardware-specific characteristics of the target platform. Beyond just considering gradient magnitudes, the system must evaluate the actual execution cost of updating specific layers on the deployed hardware. Some parameters might show high contribution scores but prove expensive to update on certain architectures, requiring a more nuanced selection strategy that balances statistical utility with runtime efficiency.

Despite these tradeoffs, task-adaptive sparse updates provide a powerful mechanism to scale adaptation to diverse deployment contexts, from microcontrollers to mobile devices (Levy et al. 2023).

14.4.3.6 Adaptation Strategy Comparison

Each adaptation strategy for on-device learning offers a distinct balance between expressivity, resource efficiency, and implementation complexity. Understanding these tradeoffs is essential when designing systems for diverse deployment targets—from ultra-low-power microcontrollers to feature-rich mobile processors.

Bias-only adaptation is the most lightweight approach, updating only scalar offsets in each layer while freezing all other parameters. This significantly reduces memory requirements and computational burden, making it suitable for devices with tight memory and energy budgets. However, its limited expressivity means it is best suited to applications where the pretrained model already captures most of the relevant task features and only minor local calibration is required.

Residual adaptation, often implemented via adapter modules, introduces a small number of trainable parameters into the frozen backbone of a neural network. This allows for greater flexibility than bias-only updates, while still maintaining control over the adaptation cost. Because the backbone remains fixed, training can be performed efficiently and safely under constrained conditions. This method supports modular personalization across tasks and users, making it a favorable choice for mobile settings where moderate adaptation capacity is needed.

Task-adaptive sparse updates offer the greatest potential for task-specific finetuning by selectively updating only a subset of layers or parameters based on their contribution to downstream performance. While this method enables expressive local adaptation, it requires a mechanism for layer selection, through profiling, contribution analysis, or meta-training, which introduces additional complexity. Nonetheless, when deployed carefully, it allows for dynamic tradeoffs between accuracy and efficiency, particularly in systems that experience large domain shifts or evolving input conditions.

These three approaches form a spectrum of tradeoffs. Their relative suitability depends on application domain, available hardware, latency constraints, and expected distribution shift. Table 14.1 summarizes their characteristics:

Table 14.1: Comparison of model adaptation strategies.

Technique	Trainable Parameters	Memory Overhead	Expressivity	Use Case Suitability	System Requirements
Bias-Only Updates	Bias terms only	Minimal	Low	Simple personalization; low variance	Extreme memory/compute limits
Residual Adapters	Adapter modules	Moderate	Moderate to High	User-specific tuning on mobile	Mobile-class SoCs with runtime support
Sparse Layer Updates	Selective parameter subsets	Variable	High (task-adaptive)	Real-time adaptation; domain shift	Requires profiling or meta-training

14.5 Data Efficiency

On-device learning systems operate in environments where data is scarce, noisy, and highly individualized. Unlike centralized machine learning pipelines that rely on large, curated datasets, edge devices typically observe only small volumes of task-relevant data—collected incrementally over time and rarely labeled in a supervised manner (W.-Y. Chen et al. 2019). This constraint fundamentally reshapes the learning process. Algorithms must extract value from minimal supervision, generalize from sparse observations, and remain robust to distributional shift. In many cases, the available data may be insufficient to train a model from scratch or even to finetune all parameters of a pretrained network. Instead, practical on-device learning relies on data-efficient techniques: few-shot adaptation, streaming updates, memory-based replay, and compressed supervision. These approaches enable models to improve over time without requiring extensive labeled datasets or centralized aggregation, making them well-suited to mobile, wearable, and embedded platforms where data acquisition is constrained by power, storage, and privacy considerations.

14.5.1 Few-Shot and Streaming

In conventional machine learning workflows, effective training typically requires large labeled datasets, carefully curated and preprocessed to ensure sufficient diversity and balance. On-device learning, by contrast, must often proceed from only a handful of local examples—collected passively through user interaction or ambient sensing, and rarely labeled in a supervised fashion. These constraints motivate two complementary adaptation strategies: few-shot learning, in which models generalize from a small, static set of examples, and streaming adaptation, where updates occur continuously as data arrives.

Few-shot adaptation is particularly relevant when the device observes a small number of labeled or weakly labeled instances for a new task or user condition (Yaqing Wang et al. 2020). In such settings, it is often infeasible to perform full finetuning of all model parameters without overfitting. Instead, methods such as bias-only updates, adapter modules, or prototype-based classification are employed to make use of limited data while minimizing capacity for memorization. Let $D = \{(x_i, y_i)\}_{i=1}^{K}$ denote a K-shot dataset of labeled examples collected on-device. The goal is to update the model parameters θ to improve task performance under constraints such as:

- Limited number of gradient steps: $T \ll 100$
- Constrained memory footprint: $\|\theta_{\text{updated}}\| \ll \|\theta\|$
- Preservation of prior task knowledge (to avoid catastrophic forgetting)

Keyword spotting (KWS) systems offer a concrete example of few-shot adaptation in a real-world, on-device deployment (Warden 2018). These models are used to detect fixed phrases, including phrases like "Hey Siri" or "OK Google", with low latency and high reliability. A typical KWS model consists of a pretrained acoustic encoder (e.g., a small convolutional or recurrent network that transforms input audio into an embedding space) followed by a lightweight classifier. In commercial systems, the encoder is trained centrally using thousands of hours of labeled speech across multiple languages and speakers. However, supporting custom wake words (e.g., "Hey Jarvis") or adapting to underrepresented accents and dialects is often infeasible via centralized training due to data scarcity and privacy concerns.

Few-shot adaptation solves this problem by finetuning only the output classifier or a small subset of parameters, including bias terms, using just a few example utterances collected directly on the device. For example, a user might provide 5–10 recordings of their custom wake word. These samples are then used to update the model locally, while the main encoder remains frozen to preserve generalization and reduce memory overhead. This enables personalization without requiring additional labeled data or transmitting private audio to the cloud.

Such an approach is not only computationally efficient, but also aligned with privacy-preserving design principles. Because only the output layer is updated, often involving a simple gradient step or prototype computation, the total memory footprint and runtime compute are compatible with mobile-class devices or even microcontrollers. This makes KWS a canonical case study for few-shot learning at the edge, where the system must operate under tight constraints while delivering user-specific performance.

Beyond static few-shot learning, many on-device scenarios benefit from streaming adaptation, where models must learn incrementally as new data arrives (Hayes et al. 2020). Streaming adaptation generalizes this idea to continuous, asynchronous settings where data arrives incrementally over time. Let $\{x_t\}_{t=1}^{\infty}$ represent a stream of observations. In streaming settings, the model must update itself after observing each new input, typically without access to prior data, and under bounded memory and compute. The model update can be written generically as:

$$\theta_{t+1} = \theta_t - \eta_t \nabla \mathcal{L}(x_t; \theta_t)$$

where η_t is the learning rate at time t. This form of adaptation is sensitive to noise and drift in the input distribution, and thus often incorporates mechanisms such as learning rate decay, meta-learned initialization, or update gating to improve stability.

Aside from KWS, practical examples of these strategies abound. In wearable health devices, a model that classifies physical activities may begin with a generic classifier and adapt to user-specific motion patterns using only a few labeled activity segments. In smart assistants, user voice profiles are finetuned over time using ongoing speech input, even when explicit supervision is unavailable. In such cases, local feedback, including correction, repetition, or downstream task success, can serve as implicit signals to guide learning.

Few-shot and streaming adaptation highlight the shift from traditional training pipelines to data-efficient, real-time learning under uncertainty. They form a foundation for more advanced memory and replay strategies, which we turn to next.

14.5.2 Experience Replay

On-device learning systems face a fundamental tension between continuous adaptation and limited data availability. One common approach to alleviating this tension is experience replay—a memory-based strategy that enables models to retrain on past examples. Originally developed in the context of reinforcement learning and continual learning, replay buffers help prevent catastrophic forgetting and stabilize training in non-stationary environments.

Unlike server-side replay strategies that rely on large datasets and extensive compute, on-device replay must operate with extremely limited capacity, often with tens or hundreds of samples, and must avoid interfering with user experience (Rolnick et al. 2019). Buffers may store only compressed features or distilled summaries, and updates must occur opportunistically (e.g., during idle cycles or charging). These system-level constraints reshape how replay is implemented and evaluated in the context of embedded ML.

Let $\mathcal{M}$ represent a memory buffer that retains a fixed-size subset of training examples. At time step t, the model receives a new data point (x_t, y_t) and appends it to $\mathcal{M}$. A replay-based update then samples a batch $\{(x_i, y_i)\}_{i=1}^{k}$ from $\mathcal{M}$ and applies a gradient step:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \left[\frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(x_i, y_i; \theta_t) \right]$$

where θ_t are the model parameters, η is the learning rate, and $\mathcal{L}$ is the loss function. Over time, this replay mechanism allows the model to reinforce prior knowledge while incorporating new information.

A practical on-device implementation might use a ring buffer[13] to store a small set of compressed feature vectors rather than full input examples. The pseudocode as shown in Listing 14.4 illustrates a minimal replay buffer designed for constrained environments.

This implementation maintains a fixed-capacity cyclic buffer, storing compressed representations (e.g., last-layer embeddings) and associated labels. Such buffers are useful for replaying adaptation updates without violating memory or energy budgets.

In TinyML applications, experience replay has been applied to problems such as gesture recognition, where devices must continuously improve predictions

[13] Ring Buffer: A circular buffer that efficiently manages data by overwriting old entries with new ones as space requires.

Listing 14.4: Minimal replay buffer implementation

```python
# Replay Buffer Techniques
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.index = 0

    def store(self, feature_vec, label):
        if len(self.buffer) < self.capacity:
            self.buffer.append((feature_vec, label))
        else:
            self.buffer[self.index] = (feature_vec, label)
        self.index = (self.index + 1) % self.capacity

    def sample(self, k):
    return random.sample(
        self.buffer,
        min(k, len(self.buffer))
    )
```

while observing a small number of events per day. Instead of training directly on the streaming data, the device stores representative feature vectors from recent gestures and uses them to finetune classification boundaries periodically. Similarly, in on-device keyword spotting, replaying past utterances can improve wake-word detection accuracy without the need to transmit audio data off-device.

While experience replay improves stability in data-sparse or non-stationary environments, it introduces several tradeoffs. Storing raw inputs may breach privacy constraints or exceed storage budgets, especially in vision and audio applications. Replaying from feature vectors reduces memory usage but may limit the richness of gradients for upstream layers. Write cycles to persistent flash memory, which are frequently necessary for long-term storage on embedded devices, can also raise wear-leveling concerns[14]. These constraints require careful co-design of memory usage policies, replay frequency, and feature selection strategies, particularly in continuous deployment scenarios.

14.5.3 Data Compression

In many on-device learning scenarios, the raw training data may be too large, noisy, or redundant to store and process effectively. This motivates the use of compressed data representations, where the original inputs are transformed into lower-dimensional embeddings or compact encodings that preserve salient information while minimizing memory and compute costs.

Compressed representations serve two complementary goals. First, they reduce the footprint of stored data, allowing devices to maintain longer histories

[14] Wear leveling is a technique used in flash memory management to distribute data writes evenly across the memory, prolonging lifespan.

or replay buffers under tight memory budgets (Sanh et al. 2019). Second, they simplify the learning task by projecting raw inputs into more structured feature spaces, often learned via pretraining or meta-learning, in which efficient adaptation is possible with minimal supervision.

One common approach is to encode data points using a pretrained feature extractor and discard the original high-dimensional input. For example, an image x_i might be passed through a convolutional neural network (CNN) to produce an embedding vector $z_i = f(x_i)$, where $f(\cdot)$ is a fixed feature encoder. This embedding captures visual structure (e.g., shape, texture, or spatial layout) in a compact representation, usually ranging from 64 to 512 dimensions, suitable for lightweight downstream adaptation.

Mathematically, training can proceed over compressed samples (z_i, y_i) using a lightweight decoder or projection head. Let θ represent the trainable parameters of this decoder model, which is typically a small neural network that maps from compressed representations to output predictions. As each example is presented, the model parameters are updated using gradient descent:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(g(z_i; \theta), y_i)$$

Here:

- z_i is the compressed representation of the i-th input,
- y_i is the corresponding label or supervision signal,
- $g(z_i; \theta)$ is the decoder's prediction,
- $\mathcal{L}$ is the loss function measuring prediction error,
- η is the learning rate, and
- ∇_θ denotes the gradient with respect to the parameters θ.

This formulation highlights how only a compact decoder model, which has the parameter set θ, needs to be trained, making the learning process feasible even when memory and compute are limited.

Advanced approaches go beyond fixed encoders by learning discrete or sparse dictionaries that represent data using low-rank or sparse coefficient matrices. For instance, a dataset of sensor traces can be factorized as $X \approx DC$, where D is a dictionary of basis patterns and C is a block-sparse coefficient matrix indicating which patterns are active in each example. By updating only a small number of dictionary atoms or coefficients, the model can adapt with minimal overhead.

Compressed representations are particularly useful in privacy-sensitive settings, as they allow raw data to be discarded or obfuscated after encoding. Furthermore, compression acts as an implicit regularizer, smoothing the learning process and mitigating overfitting when only a few training examples are available.

In practice, these strategies have been applied in domains such as keyword spotting, where raw audio signals are first transformed into Mel-frequency cepstral coefficients (MFCCs)—a compact, lossy representation of the power spectrum of speech. These MFCC vectors serve as compressed inputs for downstream models, enabling local adaptation using only a few kilobytes of memory. Instead of storing raw audio waveforms, which are large and computationally expensive to process, devices store and learn from these compressed feature vectors directly. Similarly, in low-power computer vision systems, embeddings extracted from lightweight CNNs are retained and reused for few-shot learning. These examples illustrate how representation learning and compression serve as foundational tools for scaling on-device learning to memory- and bandwidth-constrained environments.

14.5.4 Tradeoffs Summary

Each of the techniques introduced in this section, few-shot learning, experience replay, and compressed data representations, offers a strategy for adapting models on-device when data is scarce or streaming. However, they operate under different assumptions and constraints, and their effectiveness depends on system-level factors such as memory capacity, data availability, task structure, and privacy requirements.

Few-shot adaptation excels when a small but informative set of labeled examples is available, especially when personalization or rapid task-specific tuning is required. It minimizes compute and data needs, but its effectiveness hinges on the quality of pretrained representations and the alignment between the initial model and the local task.

Experience replay addresses continual adaptation by mitigating forgetting and improving stability, especially in non-stationary environments. It enables reuse of past data, but requires memory to store examples and compute cycles for periodic updates. Replay buffers may also raise privacy or longevity concerns, especially on devices with limited storage or flash write cycles.

Compressed data representations reduce the footprint of learning by transforming raw data into compact feature spaces. This approach supports longer retention of experience and efficient finetuning, particularly when only lightweight heads are trainable. However, compression can introduce information loss, and fixed encoders may fail to capture task-relevant variability if they are not well-aligned with deployment conditions. Table 14.2 summarizes key tradeoffs:

Table 14.2: Summary of on-device learning techniques.

Technique	Data Requirements	Memory/Compute Overhead	Use Case Fit
Few-Shot Adaptation	Small labeled set (K-shots)	Low	Personalization, quick on-device finetuning
Experience Replay	Streaming data	Moderate (buffer & update)	Non-stationary data, stability under drift
Compressed Representations	Unlabeled or encoded data	Low to Moderate	Memory-limited devices, privacy-sensitive contexts

In practice, these methods are not mutually exclusive. Many real-world systems combine them to achieve robust, efficient adaptation. For example, a keyword spotting system may use compressed audio features (e.g., MFCCs), finetune a few parameters from a small support set, and maintain a replay buffer of past embeddings for continual refinement.

Together, these strategies embody the core challenge of on-device learning: achieving reliable model improvement under persistent constraints on data, compute, and memory.

14.6 Federated Learning

On-device learning enables models to adapt locally using data generated on the device, but doing so in isolation limits a system's ability to generalize across users and tasks. In many applications, learning must occur not just within a single device, but across a fleet of heterogeneous, intermittently connected systems. This calls for a distributed coordination framework that supports collective model improvement without violating the constraints of privacy, limited connectivity, and device autonomy. Federated learning (FL) is one such framework.

> ℹ Definition of Federated Learning
>
> **Federated Learning** is a *decentralized machine learning approach* in which training occurs across a population of distributed devices, each using its *private, locally collected data*. Rather than transmitting raw data to a central server, devices share only *model updates*, including gradients and weight changes, which are then aggregated to improve a shared global model. This approach *preserves data privacy* while enabling *collective intelligence across diverse environments*. As federated learning matures, it integrates *privacy-enhancing technologies, communication-efficient protocols, and personalization strategies*, making it foundational for scalable, privacy-conscious ML systems.

To better understand the role of federated learning, it is useful to contrast it with other learning paradigms. Figure 14.7 illustrates the distinction between offline learning, on-device learning, and federated learning. In traditional offline learning, all data is collected and processed centrally. The model is trained in the cloud using curated datasets and is then deployed to edge devices without further adaptation. In contrast, on-device learning enables local model adaptation using data generated on the device itself, supporting personalization but in isolation—without sharing insights across users. Federated learning bridges these two extremes by enabling localized training while coordinating updates globally. It retains data privacy by keeping raw data local, yet benefits from distributed model improvements by aggregating updates from many devices.

This section explores the principles and practical considerations of federated learning in the context of mobile and embedded systems. It begins by outlining the canonical FL protocols and their system implications. It then discusses device participation constraints, communication-efficient update mechanisms, and strategies for personalized learning. Throughout, the emphasis remains on how federated methods can extend the reach of on-device learning by enabling distributed model training across diverse and resource-constrained hardware platforms.

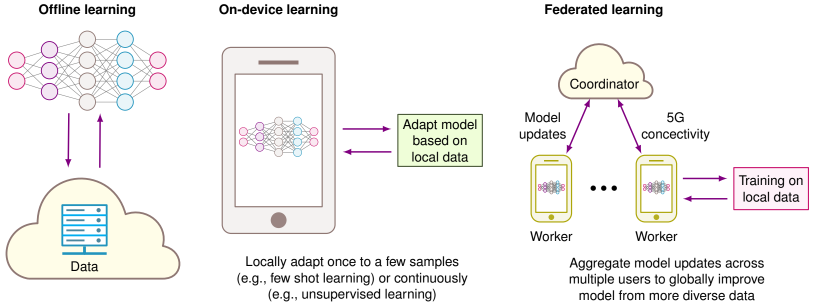

Offline learning
Data
On-device learning
Adapt model based on local data
Locally adapt once to a few samples (e.g., few shot learning) or continuously (e.g., unsupervised learning)
Federated learning
Coordinator
Model updates
5G concectivity
Worker
Worker
Training on local data
Aggregate model updates across multiple users to globally improve model from more diverse data

Figure 14.7: A comparison of learning paradigms: Offline learning occurs centrally with all data aggregated in the cloud. On-device learning adapts models locally based on user data but does not share information across users. Federated learning combines local adaptation with global coordination by aggregating model updates without sharing raw data, enabling privacy-preserving collective improvement.

14.6.1 Federated Learning Motivation

Federated learning (FL) is a decentralized paradigm for training machine learning models across a population of devices without transferring raw data to a central server (H. B. McMahan et al. 2017). Unlike traditional centralized training pipelines, which require aggregating all training data in a single location, federated learning distributes the training process itself. Each participating device computes updates based on its local data and contributes to a global model through an aggregation protocol, typically coordinated by a central server. This shift in training architecture aligns closely with the needs of mobile, edge, and embedded systems, where privacy, communication cost, and system heterogeneity impose significant constraints on centralized approaches.

The relevance of federated learning becomes apparent in several practical domains. In mobile keyboard applications, such as Google's Gboard, the system must continuously improve text prediction models based on user-specific input patterns (Hard et al. 2018). Federated learning allows the system to train on device-local keystroke data, while maintaining privacy, while still contributing to a shared model that benefits all users. Similarly, wearable health monitors often collect biometric signals that vary greatly between individuals. Training models centrally on such data would require uploading sensitive physiological traces, raising both ethical and regulatory concerns. FL mitigates these issues by enabling model updates to be computed directly on the wearable device.

In the context of smart assistants and voice interfaces, devices must adapt to individual voice profiles while minimizing false activations. Wake-word models, for instance, can be personalized locally and periodically synchronized through federated updates, avoiding the need to transmit raw voice recordings. Industrial and environmental sensors, deployed in remote locations or operating under severe bandwidth limitations, benefit from federated learning by enabling local adaptation and global coordination without constant connectivity.

These examples illustrate how federated learning bridges the gap between model improvement and system-level constraints. It enables personalization without compromising user privacy, supports learning under limited connectivity, and distributes computation across a diverse and heterogeneous device fleet. However, these benefits come with new challenges. Federated learning systems must account for client variability, communication efficiency, and the non-IID nature of local data distributions. Furthermore, they must ensure robustness to adversarial behavior and provide guarantees on model performance despite partial participation or dropout.

The remainder of this section explores the key techniques and tradeoffs that define federated learning in on-device settings. We begin by examining the core learning protocols that govern coordination across devices, and proceed to investigate strategies for scheduling, communication efficiency, and personalization.

14.6.2 Learning Protocols

Federated learning protocols define the rules and mechanisms by which devices collaborate to train a shared model. These protocols govern how local updates are computed, aggregated, and communicated, as well as how devices participate in the training process. The choice of protocol has significant implications for system performance, communication overhead, and model convergence.

In this section, we outline the core components of federated learning protocols, including local training, aggregation methods, and communication strategies. We also discuss the tradeoffs associated with different approaches and their implications for on-device learning systems.

14.6.2.1 Local Training

Local training refers to the process by which individual devices compute model updates based on their local data. This step is important in federated learning, as it allows devices to adapt the shared model to their specific contexts without transferring raw data. The local training process typically involves the following steps:

1. **Model Initialization**: Each device initializes its local model parameters, often by downloading the latest global model from the server.
2. **Local Data Sampling**: The device samples a subset of its local data for training. This data may be non-IID, meaning that it may not be uniformly distributed across devices.
3. **Local Training**: The device performs a number of training iterations on its local data, updating the model parameters based on the computed gradients.
4. **Model Update**: After local training, the device computes a model update (e.g., the difference between the updated and initial parameters) and prepares to send it to the server.
5. **Communication**: The device transmits the model update to the server, typically using a secure communication channel to protect user privacy.
6. **Model Aggregation**: The server aggregates the updates from multiple devices to produce a new global model, which is then distributed back to the participating devices.

This process is repeated iteratively, with devices periodically downloading the latest global model and performing local training. The frequency of these updates can vary based on system constraints, device availability, and communication costs.

14.6.2.2 Protocols Overview

At the heart of federated learning is a coordination mechanism that enables many devices, each having access to only a small, local dataset, to collaboratively train a shared model. This is achieved through a protocol in which client devices perform local training and periodically transmit model updates to a central server. The server aggregates these updates to refine a global model, which is then redistributed to clients for the next training round. This cyclical procedure decouples the learning process from centralized data collection, making it especially well-suited to mobile and edge environments where user data is private, bandwidth is constrained, and device participation is sporadic.

The most widely used baseline for this process is Federated Averaging (FedAvg), which has become a canonical algorithm for federated learning (H. B. McMahan et al. 2017). In FedAvg, each device trains its local copy of the model using stochastic gradient descent (SGD) on its private data. After a fixed number of local steps, each device sends its updated model parameters to the server. The server computes a weighted average of these parameters, which are weighted according to the number of data samples on each device, and updates the global model accordingly. This updated model is then sent back to the devices, completing one round of training.

Formally, let $\mathcal{D}_k$ denote the local dataset on client k, and let θ_k^t be the parameters of the model on client k at round t. Each client performs E steps of SGD on its local data, yielding an update θ_k^{t+1}. The central server then aggregates these updates as:

$$\theta^{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} \theta_k^{t+1}$$

where $n_k = |\mathcal{D}_k|$ is the number of samples on device k, $n = \sum_k n_k$ is the total number of samples across participating clients, and K is the number of active devices in the current round.

This basic structure introduces a number of design choices and tradeoffs. The number of local steps E impacts the balance between computation and communication: larger E reduces communication frequency but risks divergence if local data distributions vary too much. Similarly, the selection of participating clients affects convergence stability and fairness. In real-world deployments, not all devices are available at all times, and hardware capabilities may differ substantially, requiring robust participation scheduling and failure tolerance.

14.6.2.3 Client Scheduling

Federated learning operates under the assumption that clients, devices, which hold local data, periodically become available for participation in training rounds. However, in real-world systems, client availability is intermittent and highly variable. Devices may be turned off, disconnected from power, lacking network access, or otherwise unable to participate at any given time. As a result, client scheduling plays a central role in the effectiveness and efficiency of distributed learning.

At a baseline level, federated learning systems define eligibility criteria for participation. Devices must meet minimum requirements such as being plugged in, connected to Wi-Fi, and idle, to avoid interfering with user experience or depleting battery resources. These criteria determine which subset of the total population is considered "available" for any given training round.

Beyond these operational filters, devices also differ in their hardware capabilities, data availability, and network conditions. For example, some smartphones may contain many recent examples relevant to the current task, while others may have outdated or irrelevant data. Network bandwidth and upload speed may vary widely depending on geography and carrier infrastructure. As a result, selecting clients at random can lead to poor coverage of the underlying data distribution and unstable model convergence.

Moreover, availability-driven selection introduces participation bias: clients with favorable conditions, including frequent charging, high-end hardware, and consistent connectivity, are more likely to participate repeatedly, while others are systematically underrepresented. This can skew the resulting model toward behaviors and preferences of a privileged subset of the population, raising both fairness and generalization concerns.

To address these challenges, systems must carefully balance scheduling efficiency with client diversity. A key approach involves using stratified or quota-based sampling to ensure representative client participation across different groups. For instance, asynchronous buffer-based techniques allow participating clients to contribute model updates independently, without requiring synchronized coordination in every round (Nguyen et al. 2021). This model has been extended to incorporate staleness awareness (Rodio and Neglia 2024) and fairness mechanisms (J. Ma et al. 2024), preventing bias from over-active clients who might otherwise dominate the training process.

To address these challenges, federated learning systems implement adaptive client selection strategies. These include prioritizing clients with underrepresented data types, targeting geographies or demographics that are less frequently sampled, and using historical participation data to enforce fairness constraints. Systems may also incorporate predictive modeling to anticipate future client availability or success rates, improving training throughput.

Selected clients perform one or more local training steps on their private data and transmit their model updates to a central server. These updates are aggregated to form a new global model. Typically, this aggregation is weighted, where the contributions of each client are scaled, for example, by the number of local examples used during training, before averaging. This ensures that clients with more representative or larger datasets exert proportional influence on the global model.

These scheduling decisions directly impact system performance. They affect convergence rate, model generalization, energy consumption, and overall user experience. Poor scheduling can result in excessive stragglers, overfitting to narrow client segments, or wasted computation. As a result, client scheduling is not merely a logistical concern—it is a core component of system design in federated learning, demanding both algorithmic insight and infrastructure-level coordination.

14.6.2.4 Efficient Communication

One of the principal bottlenecks in federated learning systems is the cost of communication between edge clients and the central server. Transmitting full model weights or gradients after every training round can quickly overwhelm bandwidth and energy budgets—particularly for mobile or embedded devices operating over constrained wireless links. To address this, a range of techniques have been developed to reduce communication overhead while preserving learning efficacy.

These techniques fall into three primary categories: model compression, selective update sharing, and architectural partitioning.

Model compression methods aim to reduce the size of transmitted updates through quantization, sparsification, or subsampling. For instance, instead of sending full-precision gradients, a client may transmit 8-bit quantized updates or communicate only the top-k gradient elements with highest magnitude. These techniques significantly reduce transmission size with limited impact on convergence when applied carefully.

Selective update sharing further reduces communication by transmitting only subsets of model parameters or updates. In layer-wise selective sharing, clients may update only certain layers, typically, the final classifier or adapter modules, while keeping the majority of the backbone frozen. This reduces both upload cost and the risk of overfitting shared representations to non-representative client data.

Split models and architectural partitioning divide the model into a shared global component and a private local component. Clients train and maintain their private modules independently while synchronizing only the shared parts with the server. This allows for user-specific personalization with minimal communication and privacy leakage.

All of these approaches operate within the context of a federated aggregation protocol. A standard baseline for aggregation is Federated Averaging (FedAvg), in which the server updates the global model by computing a weighted average of the client updates received in a given round. Let $\mathcal{K}_t$ denote the set of participating clients in round t, and let θ_k^t represent the locally updated model parameters from client k. The server computes the new global model θ^{t+1} as:

$$\theta^{t+1} = \sum_{k \in \mathcal{K}_t} \frac{n_k}{n_{\mathcal{K}_t}} \theta_k^t$$

Here, n_k is the number of local training examples at client k, and $n_{\mathcal{K}_t} = \sum_{k \in \mathcal{K}_t} n_k$ is the total number of training examples across all participating clients. This data-weighted aggregation ensures that clients with more training data exert a proportionally larger influence on the global model, while also accounting for partial participation and heterogeneous data volumes.

However, communication-efficient updates can introduce tradeoffs. Compression may degrade gradient fidelity, selective updates can limit model capacity, and split architectures may complicate coordination. As a result, effective federated learning requires careful balancing of bandwidth constraints, privacy concerns, and convergence dynamics—a balance that depends heavily on the capabilities and variability of the client population.

14.6.2.5 Federated Personalization

While compression and communication strategies improve scalability, they do not address a critical limitation of the global federated learning paradigm—its inability to capture user-specific variation. In real-world deployments, devices often observe distinct and heterogeneous data distributions. A one-size-fits-all global model may underperform when applied uniformly across diverse users. This motivates the need for personalized federated learning, where local models are adapted to user-specific data without compromising the benefits of global coordination.

Let θ_k denote the model parameters on client k, and θ_{global} the aggregated global model. Traditional FL seeks to minimize a global objective:

$$\min_{\theta} \sum_{k=1}^{K} w_k \mathcal{L}_k(\theta)$$

where $\mathcal{L}_k(\theta)$ is the local loss on client k, and w_k is a weighting factor (e.g., proportional to local dataset size). However, this formulation assumes that a single model θ can serve all users well. In practice, local loss landscapes $\mathcal{L}_k$ often differ significantly across clients, reflecting non-IID data distributions and varying task requirements.

Personalization modifies this objective to allow each client to maintain its own adapted parameters θ_k, optimized with respect to both the global model and local data:

$$\min_{\theta_1,\ldots,\theta_K} \sum_{k=1}^{K} \left(\mathcal{L}_k(\theta_k) + \lambda \cdot \mathcal{R}(\theta_k, \theta_{\text{global}}) \right)$$

Here, $\mathcal{R}$ is a regularization term that penalizes deviation from the global model, and λ controls the strength of this penalty. This formulation enables local models to deviate as needed, while still benefiting from global coordination.

Real-world use cases illustrate the importance of this approach. Consider a wearable health monitor that tracks physiological signals to classify physical activities. While a global model may perform reasonably well across the population, individual users exhibit unique motion patterns, gait signatures, or sensor placements. Personalized finetuning of the final classification layer or low-rank adapters enables improved accuracy, particularly for rare or user-specific classes.

Several personalization strategies have emerged to address the tradeoffs between compute overhead, privacy, and adaptation speed. One widely used approach is local finetuning, in which each client downloads the latest global model and performs a small number of gradient steps using its private data. While this method is simple and preserves privacy, it may yield suboptimal results when the global model is poorly aligned with the client's data distribution or when the local dataset is extremely limited.

Another effective technique involves personalization layers, where the model is partitioned into a shared backbone and a lightweight, client-specific head—typically the final classification layer (Arivazhagan et al. 2019). Only the head is updated on-device, significantly reducing memory usage and training time.

This approach is particularly well-suited for scenarios in which the primary variation across clients lies in output categories or decision boundaries.

Clustered federated learning offers an alternative by grouping clients according to similarities in their data or performance characteristics, and training separate models for each cluster. This strategy can enhance accuracy within homogeneous subpopulations but introduces additional system complexity and may require exchanging metadata to determine group membership.

Finally, meta-learning approaches, such as Model-Agnostic Meta-Learning (MAML), aim to produce a global model initialization that can be quickly adapted to new tasks with just a few local updates (Finn, Abbeel, and Levine 2017). This technique is especially useful when clients have limited data or operate in environments with frequent distributional shifts. Each of these strategies reflects a different point in the tradeoff space. These strategies vary in their system implications, including compute overhead, privacy guarantees, and adaptation latency. Table 14.3 summarizes the tradeoffs.

Table 14.3: Comparison of personalization strategies in federated learning, evaluating their system-level tradeoffs across multiple design dimensions.

Strategy	Personalization Mechanism	Compute Overhead	Privacy Preservation	Adaptation Speed
Local Finetuning	Gradient descent on local loss post-aggregation	Low to Moderate	High (no data sharing)	Fast (few steps)
Personalization Layers	Split model: shared base + user-specific head	Moderate	High	Fast (train small head)
Clustered FL	Group clients by data similarity, train per group	Moderate to High	Medium (group metadata)	Medium
Meta-Learning	Train for fast adaptation across tasks/devices	High (meta-objective)	High	Very Fast (few-shot)

Selecting the appropriate personalization method depends on deployment constraints, data characteristics, and the desired balance between accuracy, privacy, and computational efficiency. In practice, hybrid approaches that combine elements of multiple strategies, including local finetuning atop a personalized head, are often employed to achieve robust performance across heterogeneous devices.

14.6.2.6 Federated Privacy

While federated learning is often motivated by privacy concerns, as it involves keeping raw data localized instead of transmitting it to a central server, the paradigm introduces its own set of security and privacy risks. Although devices do not share their raw data, the transmitted model updates (such as gradients or weight changes) can inadvertently leak information about the underlying private data. Techniques such as model inversion attacks and membership inference attacks demonstrate that adversaries may partially reconstruct or infer properties of local datasets by analyzing these updates.

To mitigate such risks, modern federated learning systems commonly employ protective measures. Secure Aggregation protocols ensure that individual model updates are encrypted and aggregated in a way that the server only observes the combined result, not any individual client's contribution. Differential Privacy techniques inject carefully calibrated noise into updates to mathematically bound the information that can be inferred about any single client's data.

While these techniques enhance privacy, they introduce additional system complexity and tradeoffs between model utility, communication cost, and robustness. A deeper exploration of these attacks, defenses, and their implications for federated and on-device learning is provided in a later security and privacy chapter.

14.7 Practical System Design

On-device learning presents opportunities for personalization, privacy preservation, and autonomous adaptation, but realizing these benefits in practice requires disciplined system design. Constraints on memory, compute, energy, and observability necessitate careful selection of adaptation mechanisms, training strategies, and deployment safeguards.

A key principle in building practical systems is to minimize the adaptation footprint. Full-model fine-tuning is typically infeasible on edge platforms, instead, localized update strategies, including bias-only optimization, residual adapters, and lightweight task-specific heads, should be prioritized. These approaches enable model specialization under resource constraints while mitigating the risks of overfitting or instability.

The feasibility of lightweight adaptation depends critically on the strength of offline pretraining (Bommasani et al. 2021). Pretrained models should encapsulate generalizable feature representations that allow efficient adaptation from limited local data. Shifting the burden of feature extraction to centralized training reduces the complexity and energy cost of on-device updates, while improving convergence stability in data-sparse environments.

Even when adaptation is lightweight, opportunistic scheduling remains essential to preserve system responsiveness and user experience. Local updates should be deferred to periods when the device is idle, connected to external power, and operating on a reliable network. Such policies minimize the impact of background training on latency, battery consumption, and thermal performance.

The sensitivity of local training artifacts necessitates careful data security measures. Replay buffers, support sets, adaptation logs, and model update metadata must be protected against unauthorized access or tampering. Lightweight encryption or hardware-backed secure storage can mitigate these risks without imposing prohibitive resource costs on edge platforms.

However, security measures alone do not guarantee model robustness. As models adapt locally, monitoring adaptation dynamics becomes critical. Lightweight validation techniques, including confidence scoring, drift detection heuristics, and shadow model evaluation, can help identify divergence early, enabling systems to trigger rollback mechanisms before severe degradation occurs (Gama et al. 2014).

Robust rollback procedures depend on retaining trusted model checkpoints. Every deployment should preserve a known-good baseline version of the model that can be restored if adaptation leads to unacceptable behavior. This principle is especially important in safety-critical and regulated domains, where failure recovery must be provable and rapid.

In decentralized or federated learning contexts, communication efficiency becomes a first-order design constraint. Compression techniques such as quantized gradient updates, sparsified parameter sets, and selective model transmission must be employed to enable scalable coordination across large, heterogeneous fleets of devices without overwhelming bandwidth or energy budgets (Konecný et al. 2016).

Moreover, when personalization is required, systems should aim for localized adaptation wherever possible. Restricting updates to lightweight components, including final classification heads or modular adapters, constrains the risk of catastrophic forgetting, reduces memory overhead, and accelerates adaptation without destabilizing core model representations.

Finally, throughout the system lifecycle, privacy and compliance requirements must be architected into adaptation pipelines. Mechanisms to support user consent, data minimization, retention limits, and the right to erasure must be considered fundamental aspects of model design, not post-hoc adjustments. Meeting regulatory obligations at scale demands that on-device learning workflows align inherently with principles of auditable autonomy.

The flowchart in Figure 14.8 summarizes key decision points in designing practical, scalable, and resilient on-device learning systems.

14.8 Challenges

While on-device learning holds significant promise for enabling adaptive, private, and efficient machine learning at the edge, its practical deployment introduces a range of challenges that extend beyond algorithm design. Unlike conventional centralized systems, where training occurs in controlled environments with uniform hardware and curated datasets, edge systems must contend with heterogeneity in devices, fragmentation in data, and the absence of centralized validation infrastructure. These factors give rise to new systems-level tradeoffs and open questions concerning reliability, safety, and maintainability. Moreover, regulatory and operational constraints complicate the deployment of self-updating models in real-world applications. This section explores these limitations, emphasizing the systemic barriers that must be addressed to make on-device learning robust, scalable, and trustworthy.

14.8.0.1 Heterogeneity

Federated and on-device learning systems must operate across a vast and diverse ecosystem of devices, ranging from smartphones and wearables to IoT sensors and microcontrollers. This heterogeneity spans multiple dimensions: hardware capabilities, software stacks, network connectivity, and power availability. Unlike cloud-based systems, where environments can be standardized

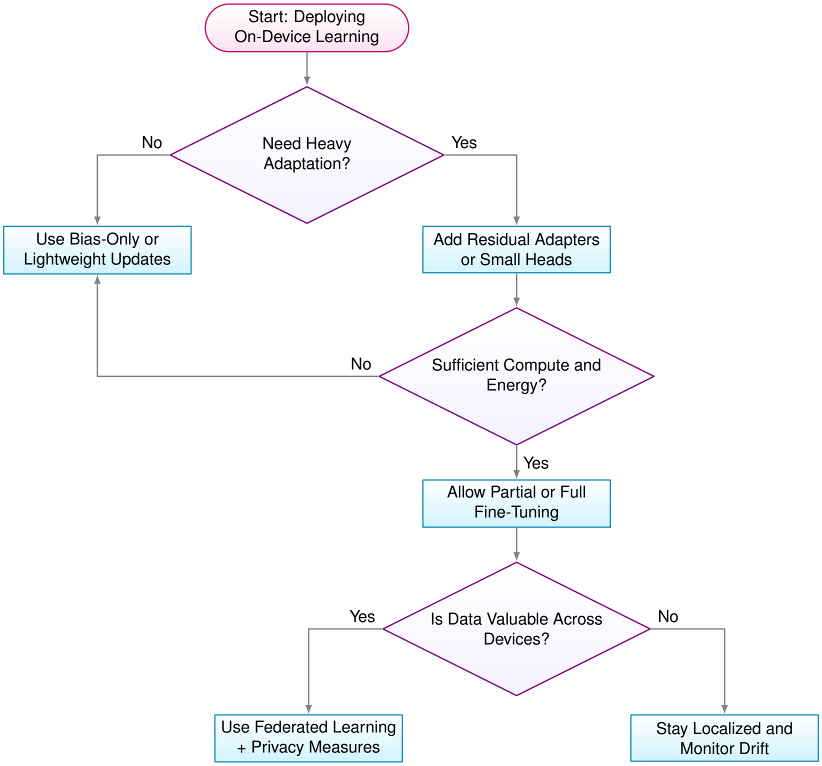

Start: Deploying On-Device Learning
Need Heavy Adaptation?
No
Yes
Use Bias-Only or Lightweight Updates
Add Residual Adapters or Small Heads
Sufficient Compute and Energy?
No
Yes
Allow Partial or Full Fine-Tuning
Is Data Valuable Across Devices?
Yes
No
Use Federated Learning + Privacy Measures
Stay Localized and Monitor Drift

Figure 14.8: Decision flowchart for designing practical on-device learning systems.

and controlled, edge deployments encounter a wide distribution of system configurations and constraints. These variations introduce significant complexity in algorithm design, resource scheduling, and model deployment.

At the hardware level, devices differ in terms of memory capacity, processor architecture (e.g., ARM Cortex-M vs. A-series), instruction set support (e.g., availability of SIMD or floating-point units), and the presence or absence of AI accelerators. Some clients may possess powerful NPUs capable of running small training loops, while others may rely solely on low-frequency CPUs with minimal RAM. These differences affect the feasible size of models, the choice of training algorithm, and the frequency of updates.

Software heterogeneity compounds the challenge. Devices may run different versions of operating systems, kernel-level drivers, and runtime libraries. Some environments support optimized ML runtimes like TensorFlow Lite Micro or ONNX Runtime Mobile, while others rely on custom inference stacks or restricted APIs. These discrepancies can lead to subtle inconsistencies in behavior, especially when models are compiled differently or when floating-point precision varies across platforms.

In addition to computational heterogeneity, devices exhibit variation in connectivity and uptime. Some are intermittently connected, plugged in only occasionally, or operate under strict bandwidth constraints. Others may have continuous power and reliable networking, but still prioritize user-facing responsiveness over background learning. These differences complicate the orchestration of coordinated learning and the scheduling of updates.

Finally, system fragmentation affects reproducibility and testing. With such a wide range of execution environments, it is difficult to ensure consistent model behavior or to debug failures reliably. This makes monitoring, validation, and rollback mechanisms more critical—but also more difficult to implement uniformly across the fleet.

Consider a federated learning deployment for mobile keyboards. A high-end smartphone might feature 8 GB of RAM, a dedicated AI accelerator, and continuous Wi-Fi access. In contrast, a budget device may have just 2 GB of RAM, no hardware acceleration, and rely on intermittent mobile data. These disparities influence how long training runs can proceed, how frequently models can be updated, and even whether training is feasible at all. To support such a range, the system must dynamically adjust training schedules, model formats, and compression strategies—ensuring equitable model improvement across users while respecting each device's limitations.

14.8.0.2 Data Fragmentation

In centralized machine learning, data can be aggregated, shuffled, and curated to approximate independent and identically distributed (IID) samples—a key assumption underlying many learning algorithms. In contrast, on-device and federated learning systems must contend with highly fragmented and non-IID data. Each device collects data specific to its user, context, and usage patterns. These data distributions are often skewed, sparse, and dynamically shifting over time.

From a statistical standpoint, the non-IID nature of on-device data leads to challenges in both optimization and generalization. Gradients computed on one device may conflict with those from another, slowing convergence or destabilizing training. Local updates can cause models to overfit to the idiosyncrasies of individual clients, reducing performance when aggregated globally. Moreover, the diversity of data across clients complicates evaluation and model validation: there is no single test set that reflects the true deployment distribution.

The fragmentation also limits the representativeness of any single client's data. Many clients may observe only a narrow slice of the input space or task distribution, making it difficult to learn robust or generalizable representations. Devices might also encounter new classes or tasks not seen during centralized pretraining, requiring mechanisms for out-of-distribution detection and continual adaptation.

These challenges demand algorithms that are robust to heterogeneity and resilient to imbalanced participation. Techniques such as personalization layers, importance weighting, and adaptive aggregation schemes attempt to mitigate these issues, but there is no universally optimal solution. The degree and nature of non-IID data varies widely across applications, making this one of the most persistent and fundamental challenges in decentralized learning.

A common example of data fragmentation arises in speech recognition systems deployed on personal assistants. Each user exhibits a unique voice profile, accent, and speaking style, which results in significant differences across local datasets. Some users may issue frequent, clearly enunciated commands, while others speak infrequently or in noisy environments. These variations cause device-specific gradients to diverge, especially when training wake-word detectors or adapting language models locally.

In federated learning deployments for virtual keyboards, the problem is further amplified. One user might primarily type in English, another in Hindi, and a third may switch fluidly between multiple languages. The resulting training data is highly non-IID—not only in language but also in vocabulary, phrasing, and typing cadence. A global model trained on aggregated updates may degrade if it fails to capture these localized differences, highlighting the need for adaptive, data-aware strategies that accommodate heterogeneity without sacrificing collective performance.

14.8.0.3 Monitoring and Validation

Unlike centralized machine learning systems, where model updates can be continuously evaluated against a held-out validation set, on-device learning introduces a fundamental shift in visibility and observability. Once deployed, models operate in highly diverse and often disconnected environments, where internal updates may proceed without external monitoring. This creates significant challenges for ensuring that model adaptation is both beneficial and safe.

A core difficulty lies in the absence of centralized validation data. In traditional workflows, models are trained and evaluated using curated datasets that serve as proxies for deployment conditions. On-device learners, by contrast, adapt in response to local inputs, which are rarely labeled and may not be systematically collected. As a result, the quality and direction of updates, whether they enhance generalization or cause drift, are difficult to assess without interfering with the user experience or violating privacy constraints.

The risk of model drift is especially pronounced in streaming settings, where continual adaptation may cause a slow degradation in performance. For instance, a voice recognition model that adapts too aggressively to background noise may eventually overfit to transient acoustic conditions, reducing accuracy on the target task. Without visibility into the evolution of model parameters or outputs, such degradations can remain undetected until they become severe.

Mitigating this problem requires mechanisms for on-device validation and update gating. One approach is to interleave adaptation steps with lightweight performance checks—using proxy objectives or self-supervised signals to approximate model confidence (Y. Deng, Mokhtari, and Ozdaglar 2021). For example, a keyword spotting system might track detection confidence across recent utterances and suspend updates if confidence consistently drops below a threshold. Alternatively, shadow evaluation can be employed, where multiple model variants are maintained on the device and evaluated in parallel on incoming data streams, allowing the system to compare the adapted model's behavior against a stable baseline.

Another strategy involves periodic checkpointing and rollback, where snapshots of the model state are saved before adaptation. If subsequent performance degrades, as determined by downstream metrics or user feedback, the system can revert to a known good state. This approach has been used in health monitoring devices, where incorrect predictions could lead to user distrust or safety concerns. However, it introduces storage and compute overhead, especially in memory-constrained environments.

In some cases, federated validation offers a partial solution. Devices can share anonymized model updates or summary statistics with a central server, which aggregates them across users to identify global patterns of drift or failure. While this preserves some degree of privacy, it introduces communication overhead and may not capture rare or user-specific failures.

Ultimately, update monitoring and validation in on-device learning require a rethinking of traditional evaluation practices. Instead of centralized test sets, systems must rely on implicit signals, runtime feedback, and conservative adaptation policies to ensure robustness. The absence of global observability is not merely a technical limitation—it reflects a deeper systems challenge in aligning local adaptation with global reliability.

14.8.0.4 Resource Management

On-device learning introduces new modes of resource contention that are not present in conventional inference-only deployments. While many edge devices are provisioned to run pretrained models efficiently, they are rarely designed with training workloads in mind. Local adaptation therefore competes for scarce resources, including compute cycles, memory bandwidth, energy, and thermal headroom, with other system processes and user-facing applications.

The most direct constraint is compute availability. Training involves additional forward and backward passes through the model, which can significantly exceed the cost of inference. Even when only a small subset of parameters is updated, for instance, in bias-only or head-only adaptation, backpropagation must still traverse the relevant layers, triggering increased instruction counts and memory traffic. On devices with shared compute units (e.g., mobile SoCs or embedded CPUs), this demand can delay interactive tasks, reduce frame rates, or impair sensor processing.

Energy consumption compounds this problem. Adaptation typically involves sustained computation over multiple input samples, which taxes battery-powered systems and may lead to rapid energy depletion. For instance, performing a single epoch of adaptation on a microcontroller-class device can consume several millijoules—an appreciable fraction of the energy budget for a duty-cycled system operating on harvested power. This necessitates careful scheduling, such that learning occurs only during idle periods, when energy reserves are high and user latency constraints are relaxed.

From a memory perspective, training incurs higher peak usage than inference, due to the need to cache intermediate activations, gradients, and optimizer state (Ji Lin et al. 2020). These requirements may exceed the static memory footprint anticipated during model deployment, particularly when adaptation involves multiple layers or gradient accumulation. In highly constrained systems, for example, systems with less than 512 KB of RAM, this may preclude certain types of adaptation altogether, unless additional optimization techniques (e.g., checkpointing or low-rank updates) are employed.

These resource demands must also be balanced against quality of service (QoS) goals. Users expect edge devices to respond reliably and consistently, regardless of whether learning is occurring in the background. Any observable degradation, including dropped audio in a wake-word detector or lag in a wearable display, can erode user trust. As such, many systems adopt opportunistic learning policies, where adaptation is suspended during foreground activity and resumed only when system load is low.

In some deployments, adaptation is further gated by cost constraints imposed by networked infrastructure. For instance, devices may offload portions of the learning workload to nearby gateways or cloudlets[15], introducing bandwidth and communication trade-offs. These hybrid models raise additional questions of task placement and scheduling: should the update occur locally, or be deferred until a high-throughput link is available?

In summary, the cost of on-device learning is not solely measured in FLOPs or memory usage. It manifests as a complex interplay of system load, user experience, energy availability, and infrastructure capacity. Addressing these challenges requires co-design across algorithmic, runtime, and hardware layers, ensuring that adaptation remains unobtrusive, efficient, and sustainable under real-world constraints.

14.8.0.5 Deployment Risks

The deployment of adaptive models on edge devices introduces challenges that extend beyond technical feasibility. In domains where compliance, auditability, and regulatory approval are necessary, including healthcare, finance, and safety-critical systems, on-device learning poses a fundamental tension between system autonomy and control.

In traditional machine learning pipelines, all model updates are centrally managed, versioned, and validated. The training data, model checkpoints, and evaluation metrics are typically recorded in reproducible workflows that support traceability. When learning occurs on the device itself, however, this visibility is lost. Each device may independently evolve its model parameters, influenced by unique local data streams that are never observed by the developer or system maintainer.

This autonomy creates a validation gap. Without access to the input data or the exact update trajectory, it becomes difficult to verify that the learned model still adheres to its original specification or performance guarantees. This is especially problematic in regulated industries, where certification depends on demonstrating that a system behaves consistently across defined operational boundaries. A device that updates itself in response to real-world usage may drift outside those bounds, triggering compliance violations without any external signal.

Moreover, the lack of centralized oversight complicates rollback and failure recovery. If a model update degrades performance, it may not be immediately detectable—particularly in offline scenarios or systems without telemetry. By the time failure is observed, the system's internal state may have diverged significantly from any known checkpoint, making diagnosis and recovery more complex than in static deployments. This necessitates robust safety mecha-

[15] Cloudlets: Smaller-scale cloud datacenters located at the edge of the internet to decrease latency for mobile and wearable devices.

nisms, such as conservative update thresholds, rollback caches, or dual-model architectures[16] that retain a verified baseline.

In addition to compliance challenges, on-device learning introduces new security vulnerabilities. Because model adaptation occurs locally and relies on device-specific, potentially untrusted data streams, adversaries may attempt to manipulate the learning process, by tampering with stored data, such as replay buffers, or by injecting poisoned examples during adaptation, to degrade model performance or introduce vulnerabilities. Furthermore, any locally stored adaptation data, such as feature embeddings or few-shot examples, must be secured against unauthorized access to prevent unintended information leakage.

Maintaining model integrity over time is particularly difficult in decentralized settings, where central monitoring and validation are limited. Autonomous updates could, without external visibility, cause models to drift into unsafe or biased states. These risks are compounded by compliance obligations such as the GDPR's right to erasure: if user data subtly influences a model through adaptation, tracking and reversing that influence becomes complex.

The security and integrity of self-adapting models, particularly at the edge, pose critical open challenges. A comprehensive treatment of these threats and corresponding mitigation strategies, including attack models and edge-specific defenses, is presented in Chapter 15: Security and Privacy.

Privacy regulations also interact with on-device learning in nontrivial ways. While local adaptation can reduce the need to transmit sensitive data, it may still require storage and processing of personal information, including sensor traces or behavioral logs, on the device itself. Depending on jurisdiction, this may invoke additional requirements for data retention, user consent, and auditability. Systems must be designed to satisfy these requirements without compromising adaptation effectiveness, which often involves encrypting stored data, enforcing retention limits, or implementing user-controlled reset mechanisms.

Lastly, the emergence of edge learning raises open questions about accountability and liability (Brakerski et al. 2022). When a model adapts autonomously, who is responsible for its behavior? If an adapted model makes a faulty decision, for instance, misdiagnosing a health condition or misinterpreting a voice command, the root cause may lie in local data drift, poor initialization, or insufficient safeguards. Without standardized mechanisms for capturing and analyzing these failure modes, responsibility may be difficult to assign, and regulatory approval harder to obtain.

Addressing these deployment and compliance risks requires new tooling, protocols, and design practices that support auditable autonomy—the ability of a system to adapt in place while still satisfying external requirements for traceability, reproducibility, and user protection. As on-device learning becomes more prevalent, these challenges will become central to both system architecture and governance frameworks.

14.8.0.6 Challenges Summary

Designing on-device learning systems involves navigating a complex landscape of technical and practical constraints. While localized adaptation enables

[16] System designs that employ two separate models, typically to enhance reliability and safety by providing a fallback.

personalization, privacy, and responsiveness, it also introduces a range of challenges that span hardware heterogeneity, data fragmentation, observability, and regulatory compliance.

System heterogeneity complicates deployment and optimization by introducing variation in compute, memory, and runtime environments. Non-IID data distributions challenge learning stability and generalization, especially when models are trained on-device without access to global context. The absence of centralized monitoring makes it difficult to validate updates or detect performance regressions, and training activity must often compete with core device functionality for energy and compute. Finally, post-deployment learning introduces complications in model governance, from auditability and rollback to privacy assurance.

These challenges are not isolated—they interact in ways that influence the viability of different adaptation strategies. Table 14.4 summarizes the primary challenges and their implications for ML systems deployed at the edge.

Table 14.4: Challenges in on-device learning and their implications for system design and deployment.

Challenge	Root Cause	System-Level Implications
System Heterogeneity	Diverse hardware, software, and toolchains	Limits portability; requires platform-specific tuning
Non-IID and Fragmented Data	Localized, user-specific data distributions	Hinders generalization; increases risk of drift
Limited Observability and Feedback	No centralized testing or logging	Makes update validation and debugging difficult
Resource Contention and Scheduling	Competing demands for memory, compute, and battery	Requires dynamic scheduling and budget-aware learning
Deployment and Compliance Risk	Learning continues post-deployment	Complicates model versioning, auditing, and rollback

14.9 Conclusion

On-device learning is a major shift in the design and operation of machine learning systems. Rather than relying exclusively on centralized training and static model deployment, this paradigm enables systems to adapt dynamically to local data and usage conditions. This shift is motivated by a confluence of factors—ranging from the need for personalization and privacy preservation to latency constraints and infrastructure efficiency. However, it also introduces a new set of challenges tied to the constrained nature of edge computing platforms.

Throughout this chapter, we explored the architectural and algorithmic strategies that make on-device learning feasible under tight compute, memory, energy, and data constraints. We began by establishing the motivation for moving learning to the edge, followed by a discussion of the system-level limitations that shape practical design choices. A core insight is that no single solution suffices across all use cases. Instead, effective on-device learning systems combine multiple techniques: minimizing the number of trainable parameters, reducing runtime costs, leveraging memory-based adaptation, and compressing data representations for efficient supervision.

We also examined federated learning as a key enabler of decentralized model refinement, particularly when coordination across many heterogeneous devices is required. While federated approaches provide strong privacy guarantees and infrastructure scalability, they introduce new concerns around client scheduling, communication efficiency, and personalization—all of which must be addressed to ensure robust real-world deployments.

Finally, we turned a critical eye toward the limitations of on-device learning, including system heterogeneity, non-IID data distributions, and the absence of reliable evaluation mechanisms in the field. These challenges underscore the importance of co-designing learning algorithms with hardware, runtime, and privacy constraints in mind.

As machine learning continues to expand into mobile, embedded, and wearable environments, the ability to adapt locally, while ensuring responsibility, efficiency, and reliability, will be essential to the next generation of intelligent systems.

14.10 Resources

ℹ Slides

- *Coming soon.*

❗ Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 15

Security & Privacy

Figure 15.1: *DALL·E 3 Prompt: An illustration on privacy and security in machine learning systems. The image shows a digital landscape with a network of interconnected nodes and data streams, symbolizing machine learning algorithms. In the foreground, there's a large lock superimposed over the network, representing privacy and security. The lock is semi-transparent, allowing the underlying network to be partially visible. The background features binary code and digital encryption symbols, emphasizing the theme of cybersecurity. The color scheme is a mix of blues, greens, and grays, suggesting a high-tech, digital environment.*

Purpose

What principles guide the protection of machine learning systems, and how do security and privacy requirements shape system architecture?

Protection mechanisms are a fundamental dimension of modern AI system design. Security considerations expose critical patterns for safeguarding data, models, and infrastructure while sustaining operational effectiveness. Implementing defensive strategies reveals inherent trade-offs between protection, performance, and usability—trade-offs that influence architectural decisions throughout the AI lifecycle. Understanding these dynamics is essential for creating trustworthy systems, grounding the principles needed to preserve privacy and defend against adversarial threats while maintaining functionality in production environments.

Learning Objectives

- Identify key security and privacy risks in machine learning systems.
- Understand how to design models with security and privacy in mind.
- Describe methods for securing model deployment and access.
- Explain strategies for monitoring and defending systems at runtime.
- Recognize the role of hardware in building trusted ML infrastructure.
- Apply a layered approach to defending machine learning systems.

15.1 Overview

Machine learning systems, like all computational systems, must be designed not only for performance and accuracy but also for security and privacy. These concerns shape the architecture and operation of ML systems across their lifecycle—from data collection and model training to deployment and user interaction. While traditional system security focuses on software vulnerabilities, network protocols, and hardware defenses, machine learning systems introduce additional and unique attack surfaces. These include threats to the data that fuels learning, the models that encode behavior, and the infrastructure that serves predictions.

Security and privacy mechanisms in ML systems serve roles analogous to trust and access control layers in classical computing. Just as operating systems enforce user permissions and protect resource boundaries, ML systems must implement controls that safeguard sensitive data, defend proprietary models, and mitigate adversarial manipulation. These mechanisms span software, hardware, and organizational layers, forming a critical foundation for system reliability and trustworthiness.

Although closely related, security and privacy address distinct aspects of protection. Security focuses on ensuring system integrity and availability in the presence of adversaries. Privacy, by contrast, emphasizes the control and protection of sensitive information, even in the absence of active attacks. These concepts often interact, but they are not interchangeable. To effectively design and evaluate defenses for ML systems, it is essential to understand how these goals differ, how they reinforce one another, and what distinct mechanisms they entail.

Security and privacy often function as complementary forces. Security prevents unauthorized access and protects system behavior, while privacy measures limit the exposure of sensitive information. Their synergy is essential: strong security supports privacy by preventing data breaches, while privacy-preserving techniques reduce the attack surface available to adversaries. However, achieving robust protection on both fronts often introduces trade-offs. Defensive mechanisms may incur computational overhead, increase system complexity, or impact usability. Designers must carefully balance these costs against protection goals, guided by an understanding of threats, system constraints, and risk tolerance.

The landscape of security and privacy challenges in ML systems continues to evolve. High-profile incidents such as model extraction attacks, data leakage from generative models, and hardware-level vulnerabilities have underscored the need for comprehensive and adaptive defenses. These solutions must address not only technical threats, but also regulatory, ethical, and operational requirements across cloud, edge, and embedded deployments.

As this chapter progresses, we will examine the threats facing machine learning systems, the defensive strategies available, and the trade-offs involved in deploying them in practice. A clear understanding of these principles is essential for building trustworthy systems that operate reliably in adversarial and privacy-sensitive environments.

15.2 Definitions and Distinctions

Security and privacy are core concerns in machine learning system design, but they are often misunderstood or conflated. While both aim to protect systems and data, they do so in different ways, address different threat models, and require distinct technical responses. For ML systems, clearly distinguishing between the two helps guide the design of robust and responsible infrastructure.

15.2.1 Security Defined

Security in machine learning focuses on defending systems from adversarial behavior. This includes protecting model parameters, training pipelines, deployment infrastructure, and data access pathways from manipulation or misuse.

> ℹ Security Definition
>
> **Security** in machine learning systems is the *protection of data, models, and infrastructure* from unauthorized access, manipulation, or disruption. It spans the *design and implementation* of defensive mechanisms that protect against data poisoning, model theft, adversarial manipulation, and system-level vulnerabilities. Security mechanisms ensure the *integrity*, *confidentiality*, and *availability* of machine learning services across development, deployment, and operational environments.

Example: A facial recognition system deployed in public transit infrastructure may be targeted with adversarial inputs that cause it to misidentify individuals or fail entirely. This is a runtime security vulnerability that threatens both accuracy and system availability.

15.2.2 Privacy Defined

Privacy focuses on limiting the exposure and misuse of sensitive information within ML systems. This includes protecting training data, inference inputs, and model outputs from leaking personal or proprietary information—even when systems operate correctly and no explicit attack is taking place.

ℹ Privacy Definition

Privacy in machine learning systems is the *protection of sensitive information* from unauthorized disclosure, inference, or misuse. It spans the *design and implementation* of methods that reduce the risk of exposing personal, proprietary, or regulated data while enabling machine learning systems to operate effectively. Privacy mechanisms help preserve *confidentiality* and *control* over data usage across development, deployment, and operational environments.

Example: A language model trained on medical transcripts may inadvertently memorize snippets of patient conversations. If a user later triggers this content through a public-facing chatbot, it represents a privacy failure—even in the absence of an attacker.

15.2.3 Security versus Privacy

Although they intersect in some areas (e.g., encrypted storage supports both), security and privacy differ in their objectives, threat models, and typical mitigation strategies. Table 15.1 below summarizes these distinctions in the context of machine learning systems.

Table 15.1: How security and privacy concerns manifest differently in machine learning systems. Security focuses on protecting against active threats that seek to manipulate or disrupt system behavior, while privacy emphasizes safeguarding sensitive information from exposure, even in benign operational contexts.

Aspect	Security	Privacy
Primary Goal	Prevent unauthorized access or disruption	Limit exposure of sensitive information
Threat Model	Adversarial actors (external or internal)	Honest-but-curious observers or passive leaks
Typical Concerns	Model theft, poisoning, evasion attacks	Data leakage, re-identification, memorization
Example Attack	Adversarial inputs cause misclassification	Model inversion reveals training data
Representative Defenses	Access control, adversarial training	Differential privacy, federated learning
Relevance to Regulation	Emphasized in cybersecurity standards	Central to data protection laws (e.g., GDPR)

15.2.4 Interactions and Trade-offs

Security and privacy are deeply interrelated but not interchangeable. A secure system helps maintain privacy by restricting unauthorized access to models and data. At the same time, privacy-preserving designs can improve security by reducing the attack surface—e.g., minimizing the retention of sensitive data reduces the risk of exposure if a system is compromised.

However, they can also be in tension. Techniques like differential privacy reduce memorization risks but may lower model utility. Encryption enhances security but may obscure transparency and auditability, complicating privacy compliance.

In machine learning systems, designers must reason about these trade-offs holistically. Systems that serve sensitive domains, including healthcare, finance, and public safety, must simultaneously protect against both misuse (security) and overexposure (privacy). Understanding the boundaries between these concerns is key to building systems that are not only performant, but trustworthy and legally compliant.

15.3 Historical Incidents

While the security of machine learning systems introduces new technical challenges, valuable lessons can be drawn from well-known security breaches across a range of computing systems. These incidents demonstrate how weaknesses in system design, in industrial control systems, connected vehicles, or consumer devices, can lead to widespread, and sometimes physical, consequences. Although the examples discussed in this section do not all involve machine learning directly, they provide critical insights into the importance of designing secure systems. These lessons apply broadly to machine learning applications deployed across cloud, edge, and embedded environments.

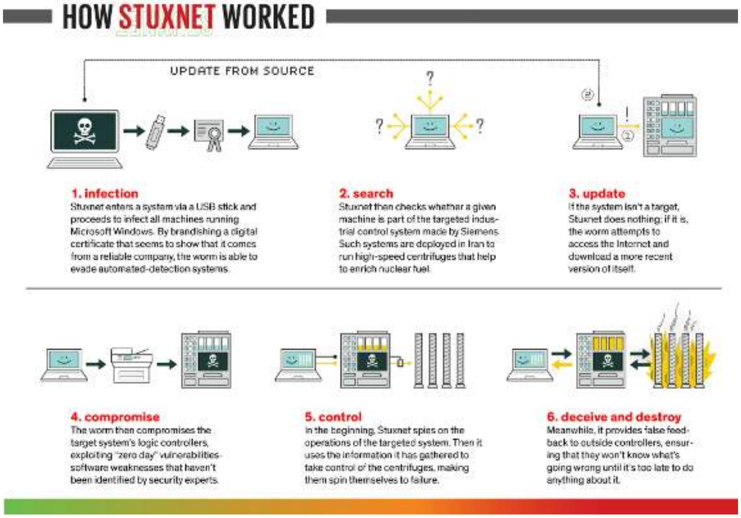
15.3.1 Stuxnet

In 2010, security researchers discovered a highly sophisticated computer worm later named Stuxnet, which targeted industrial control systems used in Iran's Natanz nuclear facility (Farwell and Rohozinski 2011). Stuxnet exploited four previously unknown "zero-day" vulnerabilities in Microsoft Windows, allowing it to spread undetected through both networked and isolated systems.

Unlike typical malware designed to steal information or perform espionage, Stuxnet was engineered to cause physical damage. Its objective was to disrupt uranium enrichment by sabotaging the centrifuges used in the process. Despite the facility being air-gapped from external networks, the malware is believed to have entered the system via an infected USB device, demonstrating how physical access can compromise even isolated environments.

Stuxnet represents a landmark in cybersecurity, revealing how malicious software can bridge the digital and physical worlds to manipulate industrial infrastructure. It specifically targeted programmable logic controllers (PLCs) responsible for automating electromechanical processes, such as controlling the speed of centrifuges. By exploiting vulnerabilities in the Windows operating system and the Siemens Step7 software used to program the PLCs, Stuxnet achieved highly targeted, real-world disruption.

While Stuxnet did not target machine learning systems directly, its relevance extends to any system where software interacts with physical processes. Machine learning is increasingly integrated into industrial control, robotics, and cyber-physical systems, making these lessons applicable to the security of modern ML deployments. Figure 15.2 illustrates the operation of Stuxnet in greater detail.



1. infection
Stuxnet enters a system via a USB stick and proceeds to infect all machines running Microsoft Windows. By brandishing a digital certificate that seems to show that it comes from a reliable company, the worm is able to evade automated-detection systems.

2. search
Stuxnet then checks whether a given machine is part of the targeted industrial control system made by Siemens. Such systems are deployed in Iran to run high-speed centrifuges that help to enrich nuclear fuel.

3. update
If the system isn't a target, Stuxnet does nothing; if it is, the worm attempts to access the Internet and download a more recent version of itself.

4. compromise
The worm then compromises the target system's logic controllers, exploiting "zero day" vulnerabilities-software weaknesses that haven't been identified by security experts.

5. control
In the beginning, Stuxnet spies on the operations of the targeted system. Then it uses the information it has gathered to take control of the centrifuges, making them spin themselves to failure.

6. deceive and destroy
Meanwhile, it provides false feedback to outside controllers, ensuring that they won't know what's going wrong until it's too late to do anything about it.

Figure 15.2: Stuxnet explained. Source: IEEE Spectrum

15.3.2 Jeep Cherokee Hack

In 2015, security researchers publicly demonstrated a remote cyberattack on a Jeep Cherokee that exposed critical vulnerabilities in automotive system design (Miller and Valasek 2015; Miller 2019). Conducted as a controlled experiment, the researchers exploited a vulnerability in the vehicle's Uconnect entertainment system, which was connected to the internet via a cellular network. By gaining remote access to this system, they were able to send commands that affected the vehicle's engine, transmission, and braking systems—without physical access to the car.

This demonstration served as a wake-up call for the automotive industry. It highlighted the risks posed by the growing connectivity of modern vehicles. Traditionally isolated automotive control systems, such as those managing steering and braking, were shown to be vulnerable when exposed through externally accessible software interfaces. The ability to remotely manipulate safety-critical functions raised serious concerns about passenger safety, regulatory oversight, and industry best practices.

Watch on YouTube
Jeep Cherokee Hack

Scan with your phone
to watch the video

! Important 8: Jeep Cherokee Hack

Watch on YouTube

The incident also led to a recall of over 1.4 million vehicles to patch the vulnerability, highlighting the need for manufacturers to prioritize cybersecurity in their designs. The National Highway Traffic Safety Administration (NHTSA) issued guidelines for automakers to improve vehicle cybersecurity, including recommendations for secure software development practices and incident response protocols.

The automotive industry has since made significant strides in addressing these vulnerabilities, but the incident serves as a cautionary tale for all sectors that rely on connected systems. As machine learning becomes more prevalent in safety-critical applications, the lessons learned from the Jeep Cherokee hack will be essential for ensuring the security and reliability of future ML deployments.

Although this incident did not involve machine learning, the architectural patterns it exposed are highly relevant to ML system security. Modern vehicles increasingly rely on machine learning for driver-assistance, navigation, and in-cabin intelligence, which include features that operate in conjunction with connected software services. This integration expands the potential attack surface if systems are not properly isolated or secured. The Jeep Cherokee hack highlights the need for defense-in-depth strategies, secure software updates, authenticated communications, and rigorous security testing—principles that apply broadly to machine learning systems deployed across automotive, industrial, and consumer environments.

As machine learning continues to be integrated into connected and safety-critical applications, the lessons from the Jeep Cherokee hack remain highly relevant. They emphasize that securing externally connected software is not just a best practice but a necessity for protecting the integrity and safety of machine learning-enabled systems.

15.3.3 Mirai Botnet

In 2016, the Mirai botnet emerged as one of the most disruptive distributed denial-of-service (DDoS) attacks in internet history (Antonakakis et al. 2017). The botnet infected thousands of networked devices, including digital cameras, DVRs, and other consumer electronics. These devices, often deployed with factory-default usernames and passwords, were easily compromised by the Mirai malware and enlisted into a large-scale attack network.

The Mirai botnet was used to overwhelm major internet infrastructure providers, disrupting access to popular online services across the United States and beyond. The scale of the attack demonstrated how vulnerable consumer and industrial devices can become a platform for widespread disruption when security is not prioritized in their design and deployment.

! Important 9: Mirai Botnet

🖥 Watch on YouTube

Watch on YouTube
Mirai Botnet
Scan with your phone
to watch the video

While the devices exploited by Mirai did not include machine learning components, the architectural patterns exposed by this incident are increasingly relevant as machine learning expands into edge computing and Internet of Things (IoT) devices. Many ML-enabled products, such as smart cameras, voice assistants, and edge analytics platforms, share similar deployment characteristics—operating on networked devices with limited hardware resources, often managed at scale.

The Mirai botnet highlights the critical importance of basic security hygiene, including secure credential management, authenticated software updates, and network access control. Without these protections, even powerful machine learning models can become part of larger attack infrastructures if deployed on insecure hardware.

As machine learning continues to move beyond centralized data centers into distributed and networked environments, the lessons from the Mirai botnet remain highly relevant. They emphasize the need for secure device provisioning, ongoing vulnerability management, and industry-wide coordination to prevent large-scale exploitation of ML-enabled systems.

15.4 Secure Design Priorities

The historical breaches described earlier reveal how weaknesses in system design, whether in hardware, software, or network infrastructure, can lead to widespread and often physical consequences. While these incidents did not directly target machine learning systems, they offer valuable insights into architectural and operational patterns that increasingly characterize modern ML deployments. These lessons point to three overarching areas of concern: device-level security, system-level isolation and control, and protection against large-scale network exploitation.

15.4.1 Device-Level Security

The Mirai botnet exemplifies how large-scale exploitation of poorly secured devices can lead to significant disruption. This attack succeeded by exploiting common weaknesses such as default usernames and passwords, unsecured firmware update mechanisms, and unencrypted communications. While often associated with consumer-grade IoT products, these vulnerabilities are increasingly relevant to machine learning systems, particularly those deployed at the edge (Antonakakis et al. 2017).

Edge ML devices, including smart cameras, industrial controllers, and wearable health monitors, typically rely on lightweight embedded hardware like ARM-based processors running minimal operating systems. These systems are designed for low-power, distributed operation but often lack the comprehensive security features found in larger computing platforms. As these devices take on more responsibility for local data processing and real-time decision-making, they become attractive targets for remote compromise.

A compromised population of such devices can be aggregated into a botnet, similar to Mirai, and leveraged for large-scale attacks. Beyond denial-of-service threats, attackers could use these ML-enabled devices to exfiltrate sensitive data, interfere with model execution, or manipulate system outputs. Without strong device-level protections, which include secure boot processes, authenticated firmware updates, and encrypted communications, edge ML deployments remain vulnerable to being turned into platforms for broader system disruption.

15.4.2 System-Level Isolation

The Jeep Cherokee hack highlighted the risks that arise when externally connected software services are insufficiently isolated from safety-critical system functions. By exploiting a vulnerability in the vehicle's Uconnect entertainment system, researchers were able to remotely manipulate core control functions such as steering and braking. This incident demonstrated that network connectivity, if not carefully managed, can expose critical system pathways to external threats.

Machine learning systems increasingly operate in similar contexts, particularly in domains such as automotive safety, healthcare, and industrial automation. Modern vehicles, for example, integrate machine learning models for driver-assistance, autonomous navigation, and sensor fusion. These models run alongside connected software services that provide infotainment, navigation updates, and remote diagnostics. Without strong system-level isolation, attackers can exploit these externally facing services to gain access to safety-critical ML components, expanding the overall attack surface.

The automotive industry's response to the Jeep Cherokee incident, which includes large-scale recalls, over-the-air software patches, and the development of industry-wide cybersecurity standards through organizations such as Auto-ISAC and the National Highway Traffic Safety Administration (NHTSA), provides a valuable example of how industries can address emerging ML security risks.

Similar isolation principles apply to other machine learning deployments, including medical devices that analyze patient data in real time, industrial controllers that optimize manufacturing processes, and infrastructure systems that manage power grids or water supplies. Securing these systems requires architectural compartmentalization of subsystems, authenticated communication channels, and validated update mechanisms. These measures help prevent external actors from escalating access or manipulating ML-driven decision-making in safety-critical environments.

15.4.3 Large-Scale Network Exploitation

The Stuxnet attack demonstrated the ability of targeted cyber operations to cross from digital systems into the physical world, resulting in real-world disruption and damage. By exploiting software vulnerabilities in industrial control systems, the attack caused mechanical failures in uranium enrichment equipment (Farwell and Rohozinski 2011). While Stuxnet did not target machine learning systems directly, it revealed critical risks that apply broadly to cyber-physical systems—particularly those involving supply chain vulnerabilities, undisclosed (zero-day) exploits, and techniques for bypassing network isolation, such as air gaps.

As machine learning increasingly powers decision-making in manufacturing, energy management, robotics, and other operational technologies, similar risks emerge. ML-based controllers that influence physical processes, including adjusting production lines, managing industrial robots, and optimizing power distribution, represent new attack surfaces. Compromising these models or the systems that deploy them can result in physical harm, operational disruption, or strategic manipulation of critical infrastructure.

Stuxnet's sophistication highlights the potential for state-sponsored or well-resourced adversaries to target ML-driven systems as part of larger geopolitical or economic campaigns. As machine learning takes on more influential roles in controlling real-world systems, securing these deployments against both cyber and physical threats becomes essential for ensuring operational resilience and public safety.

15.4.4 Toward Secure Design

Collectively, these incidents illustrate that security must be designed into machine learning systems from the outset. Protecting such systems requires attention to multiple layers of the stack, including model-level protections to defend against attacks such as model theft, adversarial manipulation, and data leakage; data pipeline security to ensure the confidentiality, integrity, and governance of training and inference data across cloud, edge, and embedded environments; system-level isolation and access control to prevent external interfaces from compromising model execution or manipulating safety-critical outputs; secure deployment and update mechanisms to safeguard runtime environments from tampering or exploitation; and continuous monitoring and incident response capabilities to detect and recover from breaches in dynamic, distributed deployments.

These priorities reflect the lessons drawn from past incidents—emphasizing the need to protect device-level resources, isolate critical system functions, and defend against large-scale exploitation. The remainder of this chapter builds on these principles, beginning with a closer examination of threats specific to machine learning models and data. It then expands the discussion to hardware-level vulnerabilities and the unique considerations of embedded ML systems. Finally, it explores defensive strategies, including privacy-preserving techniques, secure hardware mechanisms, and system-level design practices, forming a foundation for building trustworthy machine learning systems capable of withstanding both known and emerging threats.

15.5 Threats to ML Models

Building on the lessons from historical security incidents, we now turn to threats that are specific to machine learning models. These threats span the entire ML lifecycle, ranging from training-time manipulations to inference-time evasion, and fall into three broad categories: threats to model confidentiality (e.g., model theft), threats to training integrity (e.g., data poisoning), and threats to inference robustness (e.g., adversarial examples). Each category targets different vulnerabilities and requires distinct defensive strategies.

Three primary threats stand out in this context: model theft, where adversaries steal proprietary models and the sensitive knowledge they encode; data poisoning, where attackers manipulate training data to corrupt model behavior; and adversarial attacks, where carefully crafted inputs deceive models into making incorrect predictions. Each of these threats exploits different stages of the machine learning lifecycle—from data ingestion and model training to deployment and inference.

We begin with model theft, examining how attackers extract or replicate models to undermine economic value and privacy. As shown in Figure 15.3, model theft typically targets the deployment stage of the machine learning lifecycle, where trained models are exposed through APIs, on-device engines, or serialized files. This threat sits alongside others, including data poisoning during training and adversarial attacks during inference, that together span the full pipeline from data collection to real-time prediction. Understanding the lifecycle positioning of each threat helps clarify their distinct attack surfaces and appropriate defenses.

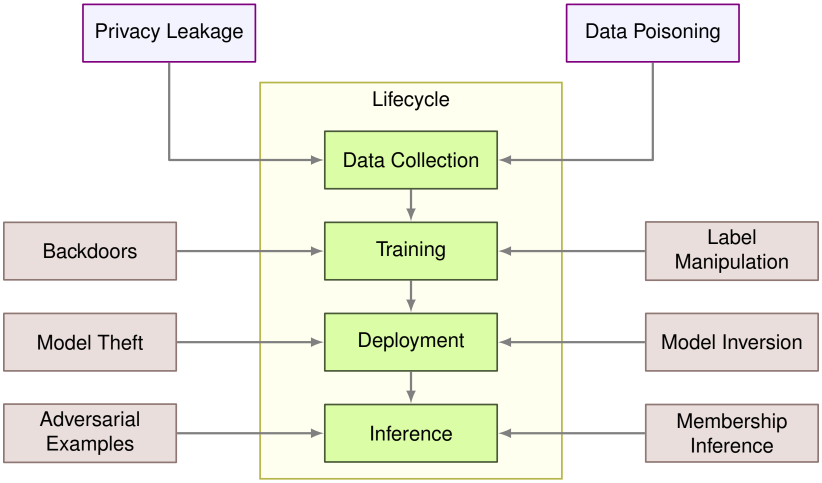


Figure 15.3: Key security and privacy threats mapped to stages of the machine learning lifecycle.

Machine learning models are not solely passive targets of attack; in some cases, they can themselves be employed as components of an attack strategy. Pretrained models, particularly large generative or discriminative networks, may be adapted to automate tasks such as adversarial example generation, phishing content synthesis, or protocol subversion. Furthermore, open-source or publicly accessible models can be fine-tuned for malicious purposes, including impersonation, surveillance, or reverse-engineering of secure systems. This dual-use potential necessitates a broader security perspective—one that considers models not only as assets to defend but also as possible instruments of attack.

15.5.1 Model Theft

Threats to model confidentiality arise when adversaries gain access to a trained model's parameters, architecture, or output behavior. These attacks can undermine the economic value of machine learning systems, enable competitors to replicate proprietary functionality, or expose private information encoded in model weights.

Such threats arise across a range of deployment settings, including public APIs, cloud-hosted services, on-device inference engines, and shared model repositories. Machine learning models may be vulnerable due to exposed interfaces, insecure serialization formats, or insufficient access controls—factors that create opportunities for unauthorized extraction or replication (Ateniese et al. 2015).

High-profile legal cases have highlighted the strategic and economic value of machine learning models. For example, former Google engineer Anthony Levandowski was accused of stealing proprietary designs from Waymo, including critical components of its autonomous vehicle technology, before founding a competing startup. Such cases illustrate the potential for insider threats to bypass technical protections and gain access to sensitive intellectual property.

The consequences of model theft extend beyond economic loss. Stolen models can be used to extract sensitive information, replicate proprietary algorithms, or enable further attacks. For instance, a competitor who obtains a stolen recommendation model from an e-commerce platform might gain insights into customer behavior, business analytics, and embedded trade secrets. This knowledge can also be used to conduct model inversion attacks, where an attacker attempts to infer private details about the model's training data (Fredrikson, Jha, and Ristenpart 2015).

In a model inversion attack, the adversary queries the model through a legitimate interface, such as a public API, and observes its outputs. By analyzing confidence scores or output probabilities, the attacker can optimize inputs to reconstruct data resembling the model's training set. For example, a facial recognition model used for secure access could be manipulated to reveal statistical properties of the employee photos on which it was trained. Similar vulnerabilities have been demonstrated in studies on the Netflix Prize dataset, where researchers were able to infer individual movie preferences from anonymized data (A. Narayanan and Shmatikov 2006).

Model theft can target two distinct objectives: extracting exact model properties, such as architecture and parameters, or replicating approximate model behavior to produce similar outputs without direct access to internal representations. Both forms of theft undermine the security and value of machine learning systems, as explored in the following subsections.

These two attack paths are illustrated in Figure 15.4. In exact model theft, the attacker gains access to the model's internal components, including serialized files, weights, and architecture definitions, and reproduces the model directly. In contrast, approximate model theft relies on observing the model's input-output behavior, typically through a public API. By repeatedly querying the model and collecting responses, the attacker trains a surrogate that mimics the original model's functionality. While the first approach compromises the model's internal design and training investment, the second threatens its predictive value and can facilitate further attacks such as adversarial example transfer or model inversion.

15.5.1.1 Exact Model Theft

Exact model property theft refers to attacks aimed at extracting the internal structure and learned parameters of a machine learning model. These attacks often target deployed models that are exposed through APIs, embedded in on-device inference engines, or shared as downloadable model files on collaboration platforms. Exploiting weak access control, insecure model packaging, or unprotected deployment interfaces, attackers can recover proprietary model assets without requiring full control of the underlying infrastructure.

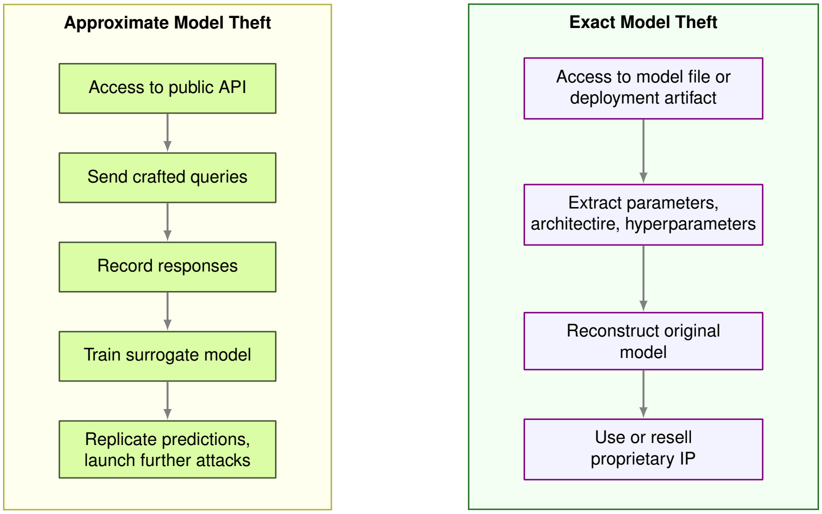

Approximate Model Theft
Access to public API
Send crafted queries
Record responses
Train surrogate model
Replicate predictions, launch further attacks
Exact Model Theft
Access to model file or deployment artifact
Extract parameters, architectire, hyperparameters
Reconstruct original model
Use or resell proprietary IP

Figure 15.4: Two primary model theft strategies: extracting internal components versus replicating external behavior.

These attacks typically seek three types of information. The first is the model's learned parameters, such as weights and biases. By extracting these parameters, attackers can replicate the model's functionality without incurring the cost of training. This replication allows them to benefit from the model's performance while bypassing the original development effort.

The second target is the model's fine-tuned hyperparameters, including training configurations such as learning rate, batch size, and regularization settings. These hyperparameters significantly influence model performance, and stealing them enables attackers to reproduce high-quality results with minimal additional experimentation.

Finally, attackers may seek to reconstruct the model's architecture. This includes the sequence and types of layers, activation functions, and connectivity patterns that define the model's behavior. Architecture theft may be accomplished through side-channel attacks, reverse engineering, or analysis of observable model behavior. Revealing the architecture not only compromises intellectual property but also gives competitors strategic insights into the design choices that provide competitive advantage.

System designers must account for these risks by securing model serialization formats, restricting access to runtime APIs, and hardening deployment pipelines. Protecting models requires a combination of software engineering practices, including access control, encryption, and obfuscation techniques, to reduce the risk of unauthorized extraction (Tramèr et al. 2016).

15.5.1.2 Approximate Model Theft

While some attackers seek to extract a model's exact internal properties, others focus on replicating its external behavior. Approximate model behavior theft refers to attacks that attempt to recreate a model's decision-making capabilities without directly accessing its parameters or architecture. Instead, attackers observe the model's inputs and outputs to build a substitute model that performs similarly on the same tasks.

This type of theft often targets models deployed as services, where the model is exposed through an API or embedded in a user-facing application. By repeatedly querying the model and recording its responses, an attacker can train their own model to mimic the behavior of the original. This process, often called model distillation or knockoff modeling, enables attackers to achieve comparable functionality without access to the original model's proprietary internals (Orekondy, Schiele, and Fritz 2019).

Attackers may evaluate the success of behavior replication in two ways. The first is by measuring the level of effectiveness of the substitute model. This involves assessing whether the cloned model achieves similar accuracy, precision, recall, or other performance metrics on benchmark tasks. By aligning the substitute's performance with that of the original, attackers can build a model that is practically indistinguishable in effectiveness, even if its internal structure differs.

The second is by testing prediction consistency. This involves checking whether the substitute model produces the same outputs as the original model when presented with the same inputs. Matching not only correct predictions but also the original model's mistakes can provide attackers with a high-fidelity reproduction of the target model's behavior. This is particularly concerning in applications such as natural language processing, where attackers might replicate sentiment analysis models to gain competitive insights or bypass proprietary systems.

Approximate behavior theft is particularly challenging to defend against in open-access deployment settings, such as public APIs or consumer-facing applications. Limiting the rate of queries, detecting automated extraction patterns, and watermarking model outputs are among the techniques that can help mitigate this risk. However, these defenses must be balanced with usability and performance considerations, especially in production environments.

One notable demonstration of approximate model theft focuses on extracting internal components of black-box language models via public APIs. In their paper, Carlini et al. (2024), researchers show how to reconstruct the final embedding projection matrix of several OpenAI models, including `ada`, `babbage`, and `gpt-3.5-turbo`, using only public API access. By exploiting the low-rank structure of the output projection layer and making carefully crafted queries, they recover the model's hidden dimensionality and replicate the weight matrix up to affine transformations.

While the attack does not reconstruct the full model, it reveals critical internal architecture parameters and sets a precedent for future, deeper extractions. This work demonstrated that even partial model theft poses risks to confidentiality and competitive advantage, especially when model behavior can be probed through rich API responses such as logit bias and log-probabilities.

Table 15.2: Model theft results from Carlini et al. (2024). The table summarizes the model sizes, number of queries required for dimension extraction, root mean square errors (RMS) for weight matrix extraction, and estimated costs based on OpenAI's API pricing.

Model	Size (Dimension Extraction)	Number of Queries	RMS (Weight Matrix Extraction)	Cost (USD)
OpenAI ada	1024 ✓	< (2 10^6)	(5 10^{-4})	$1 / $4
OpenAI babbage	2048 ✓	< (4 10^6)	(7 10^{-4})	$2 / $12
OpenAI babbage-002	1536 ✓	< (4 10^6)	Not implemented	$2 / $12
OpenAI gpt-3.5-turbo-instruct	Not disclosed	< (4 10^7)	Not implemented	$200 / ~$2,000 (estimated)
OpenAI gpt-3.5-turbo-1106	Not disclosed	< (4 10^7)	Not implemented	$800 / ~$8,000 (estimated)

As shown in their empirical evaluation, reproduced in Table 15.2, model parameters could be extracted with root mean square errors as low as 10^{-4}, confirming that high-fidelity approximation is achievable at scale. These findings raise important implications for system design, suggesting that innocuous API features, like returning top-k logits, can serve as significant leakage vectors if not tightly controlled.

15.5.1.3 Case Study: Tesla IP Theft

In 2018, Tesla filed a lawsuit against the self-driving car startup Zoox, alleging that former Tesla employees had stolen proprietary data and trade secrets related to Tesla's autonomous driving technology. According to the lawsuit, several employees transferred over 10 gigabytes of confidential files, including machine learning models and source code, before leaving Tesla to join Zoox.

Among the stolen materials was a key image recognition model used for object detection in Tesla's self-driving system. By obtaining this model, Zoox could have bypassed years of research and development, giving the company a competitive advantage. Beyond the economic implications, there were concerns that the stolen model could expose Tesla to further security risks, such as model inversion attacks aimed at extracting sensitive data from the model's training set.

The Zoox employees denied any wrongdoing, and the case was ultimately settled out of court. Nevertheless, the incident highlights the real-world risks of model theft, particularly in industries where machine learning models represent significant intellectual property. The theft of models not only undermines competitive advantage but also raises broader concerns about privacy, safety, and the potential for downstream exploitation.

This case demonstrates that model theft is not limited to theoretical attacks conducted over APIs or public interfaces. Insider threats, supply chain vulnerabilities, and unauthorized access to development infrastructure pose equally serious risks to machine learning systems deployed in commercial environments.

15.5.2 Data Poisoning

Training integrity threats stem from the manipulation of data used to train machine learning models. These attacks aim to corrupt the learning process by introducing examples that appear benign but induce harmful or biased behavior in the final model.

Data poisoning attacks are a prominent example, in which adversaries inject carefully crafted data points into the training set to influence model behavior in targeted or systemic ways (Biggio, Nelson, and Laskov 2012). Poisoned data may cause a model to make incorrect predictions, degrade its generalization ability, or embed failure modes that remain dormant until triggered post-deployment.

Data poisoning is a security threat because it involves intentional manipulation of the training data by an adversary, with the goal of embedding vulnerabilities or subverting model behavior. These attacks are especially concerning in applications where models retrain on data collected from external sources, including user interactions, crowdsourced annotations, and online scraping, since attackers can inject poisoned data without direct access to the training pipeline. Even in more controlled settings, poisoning may occur through compromised data storage, insider manipulation, or insecure data transfer processes.

From a security perspective, poisoning attacks vary depending on the attacker's level of access and knowledge. In white-box scenarios, the adversary may have detailed insight into the model architecture or training process, enabling more precise manipulation. In contrast, black-box or limited-access attacks exploit open data submission channels or indirect injection vectors. Poisoning can target different stages of the ML pipeline, ranging from data collection and preprocessing to labeling and storage, making the attack surface both broad and system-dependent.

Poisoning attacks typically follow a three-stage process. First, the attacker injects malicious data into the training set. These examples are often designed to appear legitimate but introduce subtle distortions that alter the model's learning process. Second, the model trains on this compromised data, embedding the attacker's intended behavior. Finally, once the model is deployed, the attacker may exploit the altered behavior to cause mispredictions, bypass safety checks, or degrade overall reliability.

Formally, data poisoning can be viewed as a bilevel optimization problem, where the attacker seeks to select poisoning data D_p that maximizes the model's loss on a validation or target dataset D_{test}. Let D represent the original training data. The attacker's obj

$$\max_{D_p} \mathcal{L}(f_{D \cup D_p}, D_{\text{test}})$$

where $f_{D \cup D_p}$ is the model trained on the combined dataset. For targeted attacks, this objective may focus on specific inputs x_t and target labels y_t:

$$\max_{D_p} \mathcal{L}(f_{D \cup D_p}, x_t, y_t)$$

This formulation captures the adversary's goal of introducing carefully crafted data points to manipulate the model's decision boundaries.

For example, consider a traffic sign classification model trained to distinguish between stop signs and speed limit signs. An attacker might inject a small number of stop sign images labeled as speed limit signs into the training data. The attacker's goal is to subtly shift the model's decision boundary so that future stop signs are misclassified as speed limit signs. In this case, the poisoning data D_p consists of mislabeled stop sign images, and the attacker's objective is to maximize the misclassification of legitimate stop signs x_t as speed limit signs y_t, following the targeted attack formulation above. Even if the model performs well on other types of signs, the poisoned training process creates a predictable and exploitable vulnerability.

Data poisoning attacks can be classified based on their objectives and scope of impact. Availability attacks degrade overall model performance by introducing noise or label flips that reduce accuracy across tasks. Targeted attacks manipulate a specific input or class, leaving general performance intact but causing consistent misclassification in select cases. Backdoor attacks embed hidden triggers, which are often imperceptible patterns, that elicit malicious behavior only when the trigger is present. Subpopulation attacks degrade performance on a specific group defined by shared features, making them particularly dangerous in fairness-sensitive applications.

A notable real-world example of a targeted poisoning attack was demonstrated against Perspective, an online toxicity detection model (Hosseini et al. 2017). By injecting synthetically generated toxic comments with subtle misspellings and grammatical errors into the model's training set, researchers degraded its ability to detect harmful content. After retraining, the poisoned model exhibited a significantly higher false negative rate, allowing offensive language to bypass filters. This case illustrates how poisoned data can exploit feedback loops in systems that rely on user-generated input, leading to reduced effectiveness over time and creating long-term vulnerabilities in content moderation pipelines.

Mitigating data poisoning threats requires end-to-end security of the data pipeline, encompassing collection, storage, labeling, and training. Preventative measures include input validation checks, integrity verification of training datasets, and anomaly detection to flag suspicious patterns. In parallel, robust training algorithms can limit the influence of mislabeled or manipulated data by down-weighting or filtering out anomalous instances. While no single technique guarantees immunity, combining proactive data governance, automated monitoring, and robust learning practices is essential for maintaining model integrity in real-world deployments.

15.5.3 Adversarial Attacks

Inference robustness threats occur when attackers manipulate inputs at test time to induce incorrect predictions. Unlike data poisoning, which compromises the training process, these attacks exploit vulnerabilities in the model's decision surface during inference.

A central class of such threats is adversarial attacks, where carefully constructed inputs are designed to cause incorrect predictions while remaining nearly indistinguishable from legitimate data (Szegedy et al. 2013a; Parrish et al. 2023). These attacks highlight a critical weakness in many ML models:

their sensitivity to small, targeted perturbations that can drastically alter output confidence or classification results.

The central vulnerability arises from the model's sensitivity to small, targeted perturbations. A single image, for instance, can be subtly altered, by altering only a few pixel values, such that a classifier misidentifies a stop sign as a speed limit sign. In natural language processing, specially crafted input sequences may trigger toxic or misleading outputs in a generative model, even when the prompt appears benign to a human reader (Ramesh et al. 2021; Rombach et al. 2022).

Adversarial attacks pose critical safety and security risks in domains such as autonomous driving, biometric authentication, and content moderation. Unlike data poisoning, which corrupts the model during training, adversarial attacks manipulate the model's behavior at test time, often without requiring any access to the training data or model internals. The attack surface thus shifts from upstream data pipelines to real-time interaction, demanding robust defense mechanisms capable of detecting or mitigating malicious inputs at the point of inference.

Adversarial example generation can be formally described as a constrained optimization problem, where the attacker seeks to find a minimally perturbed version of a legitimate input that maximizes the model's prediction error. Given an input x with true label y, the attacker's objective is to find a perturbed input $x' = x + \delta$ that maximizes the model's loss:

$$\max_{\delta} \mathcal{L}(f(x+\delta), y)$$

subject to the constraint:

$$\|\delta\| \leq \epsilon$$

where $f(\cdot)$ is the model, $\mathcal{L}$ is the loss function, and ϵ defines the allowed perturbation magnitude. This ensures that the perturbation remains small, often imperceptible to humans, while still leading the model to produce an incorrect output.

This optimization view underlies common adversarial strategies used in both white-box and black-box settings. A full taxonomy of attack algorithms, including gradient-based, optimization-based, and transfer-based techniques, is provided in a later chapter.

Adversarial attacks vary based on the attacker's level of access to the model. In white-box attacks, the adversary has full knowledge of the model's architecture, parameters, and training data, allowing them to craft highly effective adversarial examples. In black-box attacks, the adversary has no internal knowledge and must rely on querying the model and observing its outputs. Grey-box attacks fall between these extremes, with the adversary possessing partial information, such as access to the model architecture but not its parameters.

These attacker models can be summarized along a spectrum of knowledge levels. Table 15.3 highlights the differences in model access, data access, typical attack strategies, and common deployment scenarios. Such distinctions help characterize the practical challenges of securing ML systems across different deployment environments.

Table 15.3: Adversary knowledge spectrum.

Adversary Knowledge Level	Model Access	Training Data Access	Attack Example	Common Scenario
White-box	Full access to architecture and parameters	Full access	Crafting adversarial examples using gradients	Insider threats, open-source model reuse
Grey-box	Partial access (e.g., architecture only)	Limited or no access	Attacks based on surrogate model approximation	Known model family, unknown fine-tuning
Black-box	No internal access; only query-response view	No access	Query-based surrogate model training and transfer attacks	Public APIs, model-as-a-service deployments

A common attack strategy involves constructing a surrogate model that approximates the target model's behavior. This surrogate model is trained by querying the target model with a set of inputs $\{x_i\}$ and recording the corresponding outputs $\{f(x_i)\}$. The attacker's goal is to train a surrogate model $\hat{f}$ that minimizes the discrepancy between its predictions and those of the target model. This objective can be formulated as:

$$\min_{\hat{f}} \sum_{i=1}^{n} \ell(\hat{f}(x_i), f(x_i))$$

where ℓ is a loss function measuring the difference between the surrogate's output and the target model's output. By minimizing this loss, the attacker builds a model that behaves similarly to the target. Once trained, the surrogate model can be used to generate adversarial examples using white-box techniques. These examples often transfer to the original target model, even without internal access, making such attacks effective in black-box settings. This phenomenon, known as adversarial transferability, presents a significant challenge for defense.

Several methods have been proposed to generate adversarial examples. One notable approach leverages generative adversarial networks (GANs) (I. Goodfellow et al. 2020). In this setting, a generator network learns to produce inputs that deceive the target model, while a discriminator evaluates their effectiveness. This iterative process allows the attacker to generate sophisticated and diverse adversarial examples.

Another vector for adversarial attacks involves transfer learning pipelines. Many production systems reuse pre-trained feature extractors, fine-tuning only the final layers for specific tasks. Adversaries can exploit this structure by targeting the shared feature extractor, crafting perturbations that affect multiple downstream tasks. Headless attacks, for example, manipulate the feature extractor without requiring access to the classification head or training data (Abdelkader et al. 2020). This exposes a critical vulnerability in systems that rely on pre-trained models.

One illustrative example involves the manipulation of traffic sign recognition systems (Eykholt et al. 2017). Researchers demonstrated that placing small stickers on stop signs could cause machine learning models to misclassify them as speed limit signs. While the altered signs remained easily recognizable to humans, the model consistently misinterpreted them. Such attacks pose serious risks in applications like autonomous driving, where reliable perception is critical for safety.

Adversarial attacks highlight the need for robust defenses that go beyond improving model accuracy. Securing ML systems against adversarial threats requires runtime defenses such as input validation, anomaly detection, and monitoring for abnormal patterns during inference. Training-time robustness methods (e.g., adversarial training) complement these strategies and are discussed in more detail in a later chapter. These defenses aim to enhance model resilience against adversarial examples, ensuring that machine learning systems can operate reliably even in the presence of malicious inputs.

15.5.4 Case Study: Traffic Sign Detection Model Trickery

In 2017, researchers conducted experiments by placing small black and white stickers on stop signs (Eykholt et al. 2017). As shown in Figure 15.5, these stickers were designed to be nearly imperceptible to the human eye, yet they significantly altered the appearance of the stop sign when viewed by machine learning models. When viewed by a normal human eye, the stickers did not obscure the sign or prevent interpretability. However, when images of the stickers stop signs were fed into standard traffic sign classification ML models, they were misclassified as speed limit signs over 85% of the time.

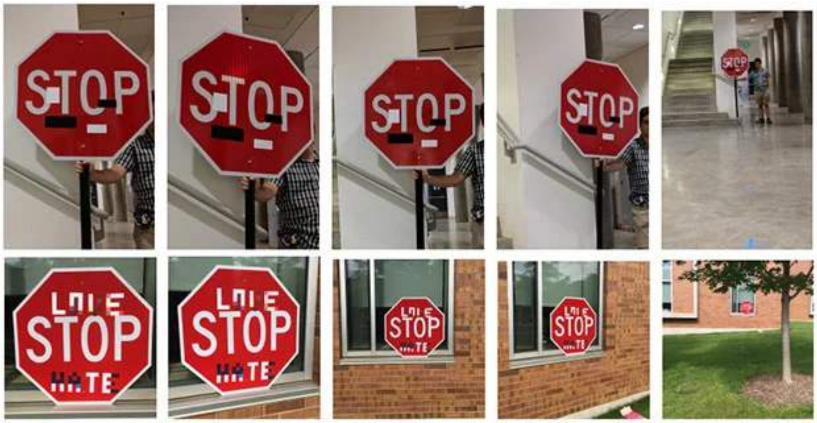


Figure 15.5: Adversarial stickers on stop signs. Source: Eykholt et al. (2017)

This demonstration showed how simple adversarial stickers could trick ML systems into misreading critical road signs. If deployed realistically, these attacks could endanger public safety, causing autonomous vehicles to misinterpret stop signs as speed limits. Researchers warned this could potentially cause dangerous rolling stops or acceleration into intersections.

This case study provides a concrete illustration of how adversarial examples exploit the pattern recognition mechanisms of ML models. By subtly altering the input data, attackers can induce incorrect predictions and pose significant risks to safety-critical applications like self-driving cars. The attack's simplicity demonstrates how even minor, imperceptible changes can lead models astray. Consequently, developers must implement robust defenses against such threats.

These threat types span different stages of the ML lifecycle and demand distinct defensive strategies. Table 15.4 below summarizes their key characteristics.

Table 15.4: Summary of threat types to ML models by lifecycle stage and attack vector.

Threat Type	Lifecycle Stage	Attack Vector	Example Impact
Model Theft	Deployment	API access, insider leaks	Stolen IP, model inversion, behavioral clone
Data Poisoning	Training	Label flipping, backdoors	Targeted misclassification, degraded accuracy
Adversarial Attacks	Inference	Input perturbation	Real-time misclassification, safety failure

The appropriate defense for a given threat depends on its type, attack vector, and where it occurs in the ML lifecycle. Figure 15.6 provides a simplified decision flow that connects common threat categories, such as model theft, data poisoning, and adversarial examples, to corresponding defensive strategies. While real-world deployments may require more nuanced or layered defenses, this flowchart serves as a conceptual guide for aligning threat models with practical mitigation techniques.

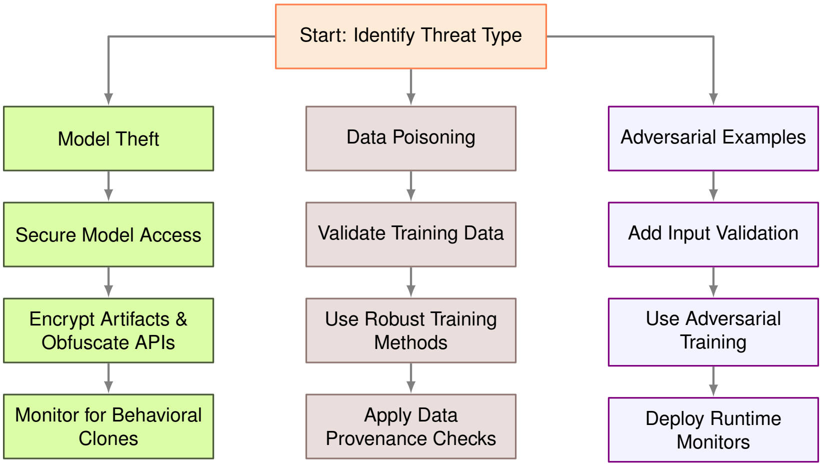


Figure 15.6: Example flow for selecting appropriate defenses based on threat type in machine learning systems.

While ML models themselves present critical attack surfaces, they ultimately run on hardware that can introduce vulnerabilities beyond the model's control. In the next section, we examine how adversaries can target the physical infrastructure that executes machine learning workloads—through hardware bugs, physical tampering, side channels, and supply chain risks.

15.6 Threats to ML Hardware

As machine learning systems move from research prototypes to large-scale, real-world deployments, their security increasingly depends on the hardware platforms they run on. Whether deployed in data centers, on edge devices, or in embedded systems, machine learning applications rely on a layered stack of processors, accelerators, memory, and communication interfaces. These hardware components, while essential for enabling efficient computation, introduce unique security risks that go beyond traditional software-based vulnerabilities.

Unlike general-purpose software systems, machine learning workflows often process high-value models and sensitive data in performance-constrained environments. This makes them attractive targets not only for software attacks but also for hardware-level exploitation. Vulnerabilities in hardware can expose models to theft, leak user data, disrupt system reliability, or allow adversaries to manipulate inference results. Because hardware operates below the software stack, such attacks can bypass conventional security mechanisms and remain difficult to detect.

These hardware threats arise from multiple sources, including design flaws in hardware architectures, physical tampering, side-channel leakage, and supply chain compromises. Together, they form a critical attack surface that must be addressed to build trustworthy machine learning systems.

Table 15.5 summarizes the major categories of hardware security threats, describing their origins, methods, and implications for machine learning system design and deployment.

Table 15.5: Threat types on hardware security.

Threat Type	Description	Relevance to ML Hardware Security
Hardware Bugs	Intrinsic flaws in hardware designs that can compromise system integrity.	Foundation of hardware vulnerability.
Physical Attacks	Direct exploitation of hardware through physical access or manipulation.	Basic and overt threat model.
Fault-injection Attacks	Induction of faults to cause errors in hardware operation, leading to potential system crashes.	Systematic manipulation leading to failure.
Side-Channel Attacks	Exploitation of leaked information from hardware operation to extract sensitive data.	Indirect attack via environmental observation.
Leaky Interfaces	Vulnerabilities arising from interfaces that expose data unintentionally.	Data exposure through communication channels.
Counterfeit Hardware	Use of unauthorized hardware components that may have security flaws.	Compounded vulnerability issues.
Supply Chain Risks	Risks introduced through the hardware lifecycle, from production to deployment.	Cumulative & multifaceted security challenges.

15.6.1 Hardware Bugs

Hardware is not immune to the pervasive issue of design flaws or bugs. Attackers can exploit these vulnerabilities to access, manipulate, or extract sensitive data, breaching the confidentiality and integrity that users and services depend on. One of the most notable examples came with the discovery of Meltdown and Spectre—two vulnerabilities in modern processors that allow malicious programs to bypass memory isolation and read the data of other applications and the operating system (Kocher et al. 2019a, 2019b).

These attacks exploit speculative execution, a performance optimization in CPUs that executes instructions out of order before safety checks are complete. While improving computational speed, this optimization inadvertently exposes sensitive data through microarchitectural side channels, such as CPU caches. The technical sophistication of these attacks highlights the difficulty of eliminating vulnerabilities even with extensive hardware validation.

Further research has revealed that these were not isolated incidents. Variants such as Foreshadow[0], ZombieLoad[1], and RIDL[2] target different microarchitectural elements, ranging from secure enclaves to CPU internal buffers, demonstrating that speculative execution flaws are a systemic hardware risk.

While these attacks were first demonstrated on general-purpose CPUs, their implications extend to machine learning accelerators and specialized hardware. ML systems often rely on heterogeneous compute platforms that combine CPUs with GPUs, TPUs, FPGAs, or custom accelerators. These components process sensitive data such as personal information, medical records, or proprietary models. Vulnerabilities in any part of this stack could expose such data to attackers.

For example, an edge device like a smart camera running a face recognition model on an accelerator could be vulnerable if the hardware lacks proper cache isolation. An attacker might exploit this weakness to extract intermediate computations, model parameters, or user data. Similar risks exist in cloud inference services, where hardware multi-tenancy increases the chances of cross-tenant data leakage.

Such vulnerabilities are particularly concerning in privacy-sensitive domains like healthcare, where ML systems routinely handle patient data. A breach could violate privacy regulations such as the Health Insurance Portability and Accountability Act (HIPAA), leading to significant legal and ethical consequences.

These examples illustrate that hardware security is not solely about preventing physical tampering. It also requires architectural safeguards to prevent data leakage through the hardware itself. As new vulnerabilities continue to emerge across processors, accelerators, and memory systems, addressing these risks requires continuous mitigation efforts—often involving performance trade-offs, especially in compute- and memory-intensive ML workloads. Proactive solutions, such as confidential computing and trusted execution environments (TEEs), offer promising architectural defenses. However, achieving robust hardware security requires attention at every stage of the system lifecycle, from design to deployment.

15.6.2 Physical Attacks

Physical tampering refers to the direct, unauthorized manipulation of computing hardware to undermine the integrity of machine learning systems. This type of attack is particularly concerning because it bypasses traditional software security defenses, directly targeting the physical components on which machine learning depends. ML systems are especially vulnerable to such attacks because they rely on hardware sensors, accelerators, and storage to process large volumes of data and produce reliable outcomes in real-world environments.

While software security measures, including encryption, authentication, and access control, protect ML systems against remote attacks, they offer little defense against adversaries with physical access to devices. Physical tampering can range from simple actions, like inserting a malicious USB device into an edge

[0] Foreshadow: A speculative execution vulnerability that targets Intel's SGX enclaves, allowing data leaks from supposedly secure memory regions.

[1] ZombieLoad: A side-channel attack that exploits Intel's CPU microarchitectural buffers to leak sensitive data processed by other applications.

[2] RIDL (Rogue In-Flight Data Load): A speculative execution attack that leaks in-flight data from CPU internal buffers, bypassing memory isolation boundaries.

server, to highly sophisticated manipulations such as embedding hardware trojans during chip manufacturing. These threats are particularly relevant for machine learning systems deployed at the edge or in physically exposed environments, where attackers may have opportunities to interfere with the hardware directly.

To understand how such attacks affect ML systems in practice, consider the example of an ML-powered drone used for environmental mapping or infrastructure inspection. The drone's navigation depends on machine learning models that process data from GPS, cameras, and inertial measurement units. If an attacker gains physical access to the drone, they could replace or modify its navigation module, embedding a hidden backdoor that alters flight behavior or reroutes data collection. Such manipulation not only compromises the system's reliability but also opens the door to misuse, such as surveillance or smuggling operations.

Physical attacks are not limited to mobility systems. Biometric access control systems, which rely on ML models to process face or fingerprint data, are also vulnerable. These systems typically use embedded hardware to capture and process biometric inputs. An attacker could physically replace a biometric sensor with a modified component designed to capture and transmit personal identification data to an unauthorized receiver. This compromises both security and user privacy, and it can enable future impersonation attacks.

In addition to tampering with external sensors, attackers may target internal hardware subsystems. For example, the sensors used in autonomous vehicles, including cameras, LiDAR, and radar, are essential for ML models that interpret the surrounding environment. A malicious actor could physically misalign or obstruct these sensors, degrading the model's perception capabilities and creating safety hazards.

Hardware trojans pose another serious risk. Malicious modifications introduced during chip fabrication or assembly can embed dormant circuits in ML accelerators or inference chips. These trojans may remain inactive under normal conditions but trigger malicious behavior when specific inputs are processed or system states are reached. Such hidden vulnerabilities can disrupt computations, leak model outputs, or degrade system performance in ways that are extremely difficult to diagnose post-deployment.

Memory subsystems are also attractive targets. Attackers with physical access to edge devices or embedded ML accelerators could manipulate memory chips to extract encrypted model parameters or training data. Fault injection techniques, including voltage manipulation and electromagnetic interference, can further degrade system reliability by corrupting model weights or forcing incorrect computations during inference.

Physical access threats extend to data center and cloud environments as well. Attackers with sufficient access could install hardware implants, such as keyloggers or data interceptors, to capture administrative credentials or monitor data streams. Such implants can provide persistent backdoor access, enabling long-term surveillance or data exfiltration from ML training and inference pipelines.

In summary, physical attacks on machine learning systems threaten both security and reliability across a wide range of deployment environments. Addressing these risks requires a combination of hardware-level protections, tamper detection mechanisms, and supply chain integrity checks. Without these safeguards, even the most secure software defenses may be undermined by vulnerabilities introduced through direct physical manipulation.

15.6.3 Fault Injection Attacks

Fault injection is a powerful class of physical attacks that deliberately disrupts hardware operations to induce errors in computation. These induced faults can compromise the integrity of machine learning models by causing them to produce incorrect outputs, degrade reliability, or leak sensitive information. For ML systems, such faults not only disrupt inference but also expose models to deeper exploitation, including reverse engineering and bypass of security protocols (Joye and Tunstall 2012).

Attackers achieve fault injection by applying precisely timed physical or electrical disturbances to the hardware while it is executing computations. Techniques such as low-voltage manipulation (Barenghi et al. 2010), power spikes (M. Hutter, Schmidt, and Plos 2009), clock glitches (Amiel, Clavier, and Tunstall 2006), electromagnetic pulses (Agrawal et al. 2007), temperature variations (S. Skorobogatov 2009), and even laser strikes (S. P. Skorobogatov and Anderson 2003) have been demonstrated to corrupt specific parts of a program's execution. These disturbances can cause effects such as bit flips, skipped instructions, or corrupted memory states, which adversaries can exploit to alter ML model behavior or extract sensitive information.

For machine learning systems, these attacks pose several concrete risks. Fault injection can degrade model accuracy, force incorrect classifications, trigger denial of service, or even leak internal model parameters. For example, attackers could inject faults into an embedded ML model running on a microcontroller, forcing it to misclassify inputs in safety-critical applications such as autonomous navigation or medical diagnostics. More sophisticated attackers may target memory or control logic to steal intellectual property, such as proprietary model weights or architecture details.

Experimental demonstrations have shown the feasibility of such attacks. One notable example is the work by Breier et al. (2018), where researchers successfully used a laser fault injection attack on a deep neural network deployed on a microcontroller. By heating specific transistors, as shown in Figure 15.7. they forced the hardware to skip execution steps, including a ReLU activation function.

This manipulation is illustrated in Figure 15.8, which shows a segment of assembly code implementing the ReLU activation function. Normally, the code compares the most significant bit (MSB) of the accumulator to zero and uses a brge (branch if greater or equal) instruction to skip the assignment if the value is non-positive. However, the fault injection suppresses the branch, causing the processor to always execute the "else" block. As a result, the neuron's output is forcibly zeroed out, regardless of the input value.

Fault injection attacks can also be combined with side-channel analysis, where attackers first observe power or timing characteristics to infer model structure or data flow. This reconnaissance allows them to target specific layers or operations,

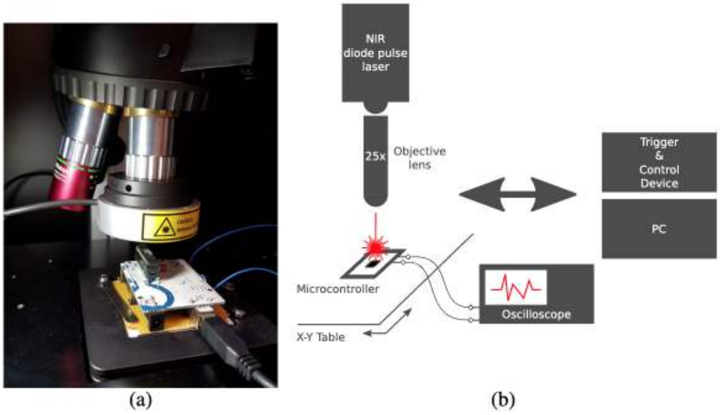


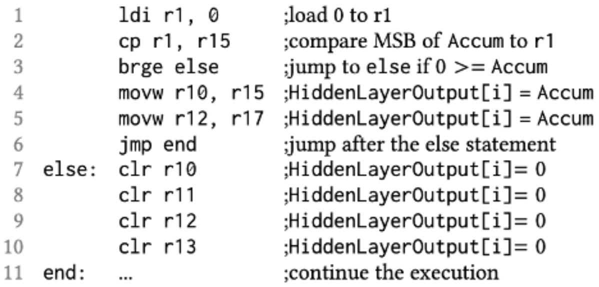
Figure 15.7: Laser fault injection attack on a microcontroller. Source: Breier et al. (2018).

such as activation functions or final decision layers, maximizing the impact of the injected faults.

Embedded and edge ML systems are particularly vulnerable because they often lack physical hardening and operate under resource constraints that limit runtime defenses. Without tamper-resistant packaging or secure hardware enclaves, attackers may gain direct access to system buses and memory, enabling precise fault manipulation. Furthermore, many embedded ML models are designed to be lightweight, leaving them with little redundancy or error correction to recover from induced faults.

Mitigating fault injection requires a multi-layered defense strategy. Physical protections, such as tamper-proof enclosures and design obfuscation, help limit physical access. Anomaly detection techniques can monitor sensor inputs or model outputs for signs of fault-induced inconsistencies (Hsiao et al. 2023). Error-correcting memories and secure firmware can reduce the likelihood of silent corruption. Techniques such as model watermarking may provide traceability if stolen models are later deployed by an adversary.

However, these protections are difficult to implement in cost- and power-constrained environments, where adding cryptographic hardware or redundancy may not be feasible. As a result, achieving resilience to fault injection



```
        ldi r1, 0       ;load 0 to r1
        cp r1, r15      ;compare MSB of Accum to r1
        brge else       ;jump to else if 0 >= Accum
        movw r10, r15   ;HiddenLayerOutput[i] = Accum
        movw r12, r17   ;HiddenLayerOutput[i] = Accum
        jmp end         ;jump after the else statement
else:   clr r10         ;HiddenLayerOutput[i]= 0
        clr r11         ;HiddenLayerOutput[i]= 0
        clr r12         ;HiddenLayerOutput[i]= 0
        clr r13         ;HiddenLayerOutput[i]= 0
end:    ...             ;continue the execution
```

Figure 15.8: Fault-injection demonstrated with assembly code. Source: Breier et al. (2018).

requires cross-layer design considerations that span electrical, firmware, software, and system architecture levels. Without such holistic design practices, ML systems deployed in the field may remain exposed to these low-cost yet highly effective physical attacks.

15.6.4 Side-Channel Attacks

Side-channel attacks constitute a class of security breaches that exploit information inadvertently revealed through the physical implementation of computing systems. In contrast to direct attacks that target software or network vulnerabilities, these attacks leverage the system's hardware characteristics, including power consumption, electromagnetic emissions, or timing behavior, to extract sensitive information.

The fundamental premise of a side-channel attack is that a device's operation can leak information through observable physical signals. Such leaks may originate from the electrical power the device consumes (Kocher, Jaffe, and Jun 1999), the electromagnetic fields it emits (Gandolfi, Mourtel, and Olivier 2001), the time it takes to complete computations, or even the acoustic noise it produces. By carefully measuring and analyzing these signals, attackers can infer internal system states or recover secret data.

Although these techniques are commonly discussed in cryptography, they are equally relevant to machine learning systems. ML models deployed on hardware accelerators, embedded devices, or edge systems often process sensitive data. Even when these models are protected by secure algorithms or encryption, their physical execution may leak side-channel signals that can be exploited by adversaries.

One of the most widely studied examples involves Advanced Encryption Standard (AES) implementations. While AES is mathematically secure, the physical process of computing its encryption functions leaks measurable signals. Techniques such as Differential Power Analysis (DPA) (Kocher et al. 2011), Differential Electromagnetic Analysis (DEMA), and Correlation Power Analysis (CPA) exploit these physical signals to recover secret keys.

A useful example of this attack technique can be seen in a power analysis of a password authentication process. Consider a device that verifies a 5-byte password—in this case, `0x61, 0x52, 0x77, 0x6A, 0x73`. During authentication, the device receives each byte sequentially over a serial interface, and its power consumption pattern reveals how the system responds as it processes these inputs.

Figure 15.9 shows the device's behavior when the correct password is entered. The red waveform captures the serial data stream, marking each byte as it is received. The blue curve records the device's power consumption over time. When the full, correct password is supplied, the power profile remains stable and consistent across all five bytes, providing a clear baseline for comparison with failed attempts.

When an incorrect password is entered, the power analysis chart changes as shown in Figure 15.10. In this case, the first three bytes (`0x61, 0x52, 0x77`) are correct, so the power patterns closely match the correct password up to that point. However, when the fourth byte (`0x42`) is processed and found to

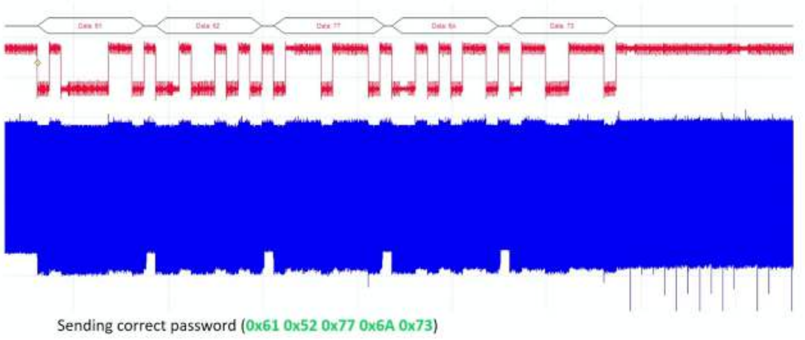


Figure 15.9: Power consumption profile of the device during normal operations with a valid 5-byte password (0x61, 0x52, 0x77, 0x6A, 0x73). The red line represents the serial data being received by the bootloader, which in this figure is receiving the correct bytes. Notice how the blue line, representing power usage during authentication, corresponds to receiving and verifying the bytes. In the next figures, this blue power consumption profile will change. Source: Colin O'Flynn.

be incorrect, the device halts authentication. This change is reflected in the sudden jump in the blue power line, indicating that the device has stopped processing and entered an error state.

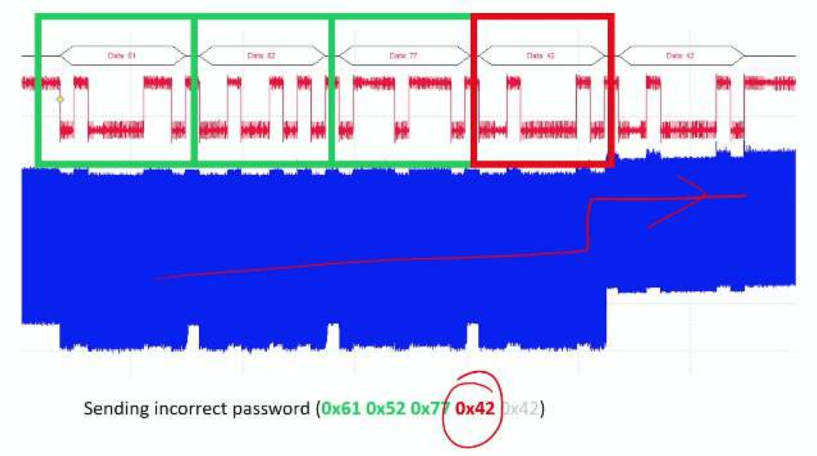


Figure 15.10: Power consumption profile of the device when an incorrect 5-byte password (0x61, 0x52, 0x77, 0x42, 0x42) is entered. The red line represents the serial data received by the bootloader, showing the input bytes being processed. The first three bytes (0x61, 0x52, 0x77) are correct and match the expected password, as indicated by the consistent blue power consumption line. However, upon processing the fourth byte (0x42), a mismatch is detected. The bootloader stops further processing, resulting in a noticeable jump in the blue power consumption line, as the device halts authentication and enters an error state. Source: Colin O'Flynn.

Figure 15.11 shows the case where the password is entirely incorrect (`0x30, 0x30, 0x30, 0x30, 0x30`). Here, the device detects the mismatch immediately after the first byte and halts processing much earlier. This is again visible in the power profile, where the blue line exhibits a sharp jump following the first byte, reflecting the device's early termination of authentication.

These examples demonstrate how attackers can exploit observable power consumption differences to reduce the search space and eventually recover secret data through brute-force analysis. For a more detailed walkthrough, Video 10 provides a step-by-step demonstration of how these attacks are performed.

Watch on YouTube
Power Attack

Scan with your phone
to watch the video

! Important 10: Power Attack

Watch on YouTube

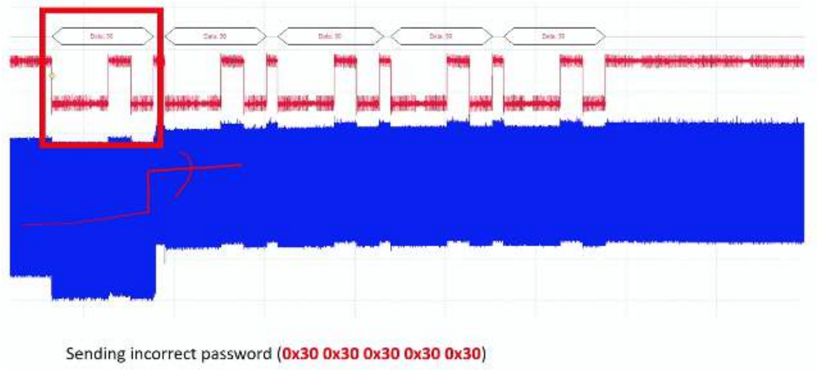


Figure 15.11: Power consumption profile of the device when an entirely incorrect password (0x30, 0x30, 0x30, 0x30, 0x30) is entered. The blue line shows a sharp jump after processing the first byte, indicating that the device has halted the authentication process. Source: Colin O'Flynn.

Such attacks are not limited to cryptographic systems. Machine learning applications face similar risks. For example, an ML-based speech recognition system processing voice commands on a local device could leak timing or power signals that reveal which commands are being processed. Even subtle acoustic or electromagnetic emissions may expose operational patterns that an adversary could exploit to infer user behavior.

Historically, side-channel attacks have been used to bypass even the most secure cryptographic systems. In the 1960s, British intelligence agency MI5 famously exploited acoustic emissions from a cipher machine in the Egyptian Embassy (Burnet and Thomas 1989). By capturing the mechanical clicks of the machine's rotors, MI5 analysts were able to dramatically reduce the complexity of breaking encrypted messages. This early example illustrates that side-channel vulnerabilities are not confined to the digital age but are rooted in the physical nature of computation.

Today, these techniques have advanced to include attacks such as keyboard eavesdropping (Asonov and Agrawal, n.d.), power analysis on cryptographic hardware (Gnad, Oboril, and Tahoori 2017), and voltage-based attacks on ML accelerators (M. Zhao and Suh 2018). Timing attacks, electromagnetic leakage, and thermal emissions continue to provide adversaries with indirect channels for observing system behavior.

Machine learning systems deployed on specialized accelerators or embedded platforms are especially at risk. Attackers may exploit side-channel signals to infer model structure, steal parameters, or reconstruct private training data. As ML becomes increasingly deployed in cloud, edge, and embedded environments, these side-channel vulnerabilities pose significant challenges to system security.

Understanding the persistence and evolution of side-channel attacks is essential for building resilient machine learning systems. By recognizing that where there is a signal, there is potential for exploitation, system designers can begin to address these risks through a combination of hardware shielding, algorithmic defenses, and operational safeguards.

15.6.5 Leaky Interfaces

Interfaces in computing systems are essential for enabling communication, diagnostics, and updates. However, these same interfaces can become significant security vulnerabilities when they unintentionally expose sensitive information or accept unverified inputs. Such leaky interfaces often go unnoticed during system design, yet they provide attackers with powerful entry points to extract data, manipulate functionality, or introduce malicious code.

A leaky interface is any access point that reveals more information than intended, often because of weak authentication, lack of encryption, or inadequate isolation. These issues have been widely demonstrated across consumer, medical, and industrial systems.

For example, many WiFi-enabled baby monitors have been found to expose unsecured remote access ports, allowing attackers to intercept live audio and video feeds from inside private homes[3]. Similarly, researchers have identified wireless vulnerabilities in pacemakers that could allow attackers to manipulate cardiac functions if exploited, raising life-threatening safety concerns[4].

A notable case involving smart lightbulbs demonstrated that accessible debug ports left on production devices leaked unencrypted WiFi credentials. This security oversight provided attackers with a pathway to infiltrate home networks without needing to bypass standard security mechanisms.[5] In the automotive domain, unsecured OBD-II diagnostic ports have allowed attackers to manipulate braking and steering functions in connected vehicles, as demonstrated in the well-known Jeep Cherokee hack.[6]

While these examples do not target machine learning systems directly, they illustrate architectural patterns that are highly relevant to ML-enabled devices. Consider a smart home security system that uses machine learning to detect user routines and automate responses. Such a system may include a maintenance or debug interface for software updates. If this interface lacks proper authentication or transmits data unencrypted, attackers on the same network could gain unauthorized access. This intrusion could expose user behavior patterns, compromise model integrity, or disable security features altogether.

Leaky interfaces in ML systems can also expose training data, model parameters, or intermediate outputs. Such exposure can enable attackers to craft adversarial examples, steal proprietary models, or reverse-engineer system behavior. Worse still, these interfaces may allow attackers to tamper with firmware, introducing malicious code that disables devices or recruits them into botnets.

Mitigating these risks requires multi-layered defenses. Technical safeguards such as strong authentication, encrypted communications, and runtime anomaly detection are essential. Organizational practices such as interface inventories, access control policies, and ongoing audits are equally important. Adopting a zero-trust architecture, where no interface is trusted by default, further reduces exposure by limiting access to only what is strictly necessary.

For designers of ML-powered systems, securing interfaces must be a first-class concern alongside algorithmic and data-centric design. Whether the system operates in the cloud, on the edge, or in embedded environments,

[3] See this report on baby monitor vulnerabilities that allowed remote attackers to eavesdrop on live feeds in private homes.

[4] Vulnerabilities in connected pacemakers raised concerns about remote manipulation of cardiac functions, as described in this medical advisory.

[5] Debug ports on consumer smart lightbulbs leaked unencrypted network credentials, as documented by Greengard (2021).

[6] The Jeep Cherokee hack demonstrated how attackers could control vehicle functions through the OBD-II port. See Miller and Valasek (2015).

failure to secure these access points risks undermining the entire system's trustworthiness.

15.6.6 Counterfeit Hardware

Machine learning systems depend on the reliability and security of the hardware on which they run. Yet, in today's globalized hardware ecosystem, the risk of counterfeit or cloned hardware has emerged as a serious threat to system integrity. Counterfeit components refer to unauthorized reproductions of genuine parts, designed to closely imitate their appearance and functionality. These components can enter machine learning systems through complex procurement and manufacturing processes that span multiple vendors and regions.

A single lapse in component sourcing can introduce counterfeit hardware into critical systems. For example, a facial recognition system deployed for secure facility access might unknowingly rely on counterfeit processors. These unauthorized components could fail to process biometric data correctly or introduce hidden vulnerabilities that allow attackers to bypass authentication controls.

The risks posed by counterfeit hardware are multifaceted. From a reliability perspective, such components often degrade faster, perform unpredictably, or fail under load due to substandard manufacturing. From a security perspective, counterfeit hardware may include hidden backdoors or malicious circuitry, providing attackers with undetectable pathways to compromise machine learning systems. A cloned network router installed in a data center, for instance, could silently intercept model predictions or user data, undermining both system security and user privacy.

Legal and regulatory risks further compound the problem. Organizations that unknowingly integrate counterfeit components into their ML systems may face serious legal consequences, including penalties for violating safety, privacy, or cybersecurity regulations. This is particularly concerning in sectors such as healthcare and finance, where compliance with industry standards is non-negotiable.

Economic pressures often incentivize sourcing from lower-cost suppliers without rigorous verification, increasing the likelihood of counterfeit parts entering production systems. Detection is especially challenging, as counterfeit components are designed to mimic legitimate ones. Identifying them may require specialized equipment or forensic analysis, making prevention far more practical than remediation.

The stakes are particularly high in machine learning applications that require high reliability and low latency, such as real-time decision-making in autonomous vehicles, industrial automation, or critical healthcare diagnostics. Hardware failure in these contexts can lead not only to system downtime but also to significant safety risks.

As machine learning continues to expand into safety-critical and high-value applications, counterfeit hardware presents a growing risk that must be recognized and addressed. Organizations must treat hardware trustworthiness as a fundamental design requirement, on par with algorithmic accuracy and data security, to ensure that ML systems can operate reliably and securely in the real world.

15.6.7 Supply Chain Risks

While counterfeit hardware presents a serious challenge, it is only one part of the larger problem of securing the global hardware supply chain. Machine learning systems are built from components that pass through complex supply networks involving design, fabrication, assembly, distribution, and integration. Each of these stages presents opportunities for tampering, substitution, or counterfeiting—often without the knowledge of those deploying the final system.

Malicious actors can exploit these vulnerabilities in various ways. A contracted manufacturer might unknowingly receive recycled electronic waste that has been relabeled as new components. A distributor might deliberately mix cloned parts into otherwise legitimate shipments. Insiders at manufacturing facilities might embed hardware Trojans that are nearly impossible to detect once the system is deployed. Advanced counterfeits can be particularly deceptive, with refurbished or repackaged components designed to pass visual inspection while concealing inferior or malicious internals.

Identifying such compromises typically requires sophisticated analysis, including micrography, X-ray screening, and functional testing. However, these methods are costly and impractical for large-scale procurement. As a result, many organizations deploy systems without fully verifying the authenticity and security of every component.

The risks extend beyond individual devices. Machine learning systems often rely on heterogeneous hardware platforms, integrating CPUs, GPUs, memory, and specialized accelerators sourced from a global supply base. Any compromise in one part of this chain can undermine the security of the entire system. These risks are further amplified when systems operate in shared or multi-tenant environments, such as cloud data centers or federated edge networks, where hardware-level isolation is critical to preventing cross-tenant attacks.

The 2018 Bloomberg Businessweek report alleging that Chinese state actors inserted spy chips into Supermicro server motherboards brought these risks to mainstream attention. While the claims remain disputed, the story underscored the industry's limited visibility into its own hardware supply chains. Companies often rely on complex, opaque manufacturing and distribution networks, leaving them vulnerable to hidden compromises. Over-reliance on single manufacturers or regions, including the semiconductor industry's reliance on TSMC, further concentrates this risk. This recognition has driven policy responses like the U.S. CHIPS and Science Act, which aims to bring semiconductor production onshore and strengthen supply chain resilience.

Securing machine learning systems requires moving beyond trust-by-default models toward zero-trust supply chain practices. This includes screening suppliers, validating component provenance, implementing tamper-evident protections, and continuously monitoring system behavior for signs of compromise.

Building fault-tolerant architectures that detect and contain failures provides an additional layer of defense.

Ultimately, supply chain risks must be treated as a first-class concern in ML system design. Trust in the computational models and data pipelines that power machine learning depends fundamentally on the trustworthiness of the hardware on which they run. Without securing the hardware foundation, even the most sophisticated models remain vulnerable to compromise.

15.6.8 Case Study: The Supermicro Hardware Security Controversy

In 2018, Bloomberg Businessweek published a widely discussed report alleging that Chinese state-sponsored actors had secretly implanted tiny surveillance chips on server motherboards manufactured by Supermicro (Robertson and Riley 2018). These compromised servers were reportedly deployed by more than 30 major companies, including Apple and Amazon. The chips, described as no larger than a grain of rice, were said to provide attackers with backdoor access to sensitive data and systems.

The allegations sparked immediate concern across the technology industry, raising questions about the security of global supply chains and the potential for state-level hardware manipulation. However, the companies named in the report publicly denied the claims. Apple, Amazon, and Supermicro stated that they had found no evidence of the alleged implants after conducting thorough internal investigations. Industry experts and government agencies also expressed skepticism, noting the lack of verifiable technical evidence presented in the report.

Despite these denials, the story had a lasting impact on how organizations and policymakers view hardware supply chain security. Whether or not the specific claims were accurate, the report highlighted the real and growing concern that hardware supply chains are difficult to fully audit and secure. It underscored how geopolitical tensions, manufacturing outsourcing, and the complexity of modern hardware ecosystems make it increasingly challenging to guarantee the integrity of hardware components.

The Supermicro case illustrates a broader truth: once a product enters a complex global supply chain, it becomes difficult to ensure that every component is free from tampering or unauthorized modification. This risk is particularly acute for machine learning systems, which depend on a wide range of hardware accelerators, memory modules, and processing units sourced from multiple vendors across the globe.

In response to these risks, both industry and government stakeholders have begun to invest in supply chain security initiatives. The U.S. government's CHIPS and Science Act is one such effort, aiming to bring semiconductor manufacturing back onshore to improve transparency and reduce dependency on foreign suppliers. While these efforts are valuable, they do not fully eliminate supply chain risks. They must be complemented by technical safeguards, such as component validation, runtime monitoring, and fault-tolerant system design.

The Supermicro controversy serves as a cautionary tale for the machine learning community. It demonstrates that hardware security cannot be taken for granted, even when working with reputable suppliers. Ensuring the integrity of

ML systems requires rigorous attention to the entire hardware lifecycle—from design and fabrication to deployment and maintenance. This case reinforces the need for organizations to adopt comprehensive supply chain security practices as a foundational element of trustworthy ML system design.

15.7 Defensive Strategies

Designing secure and privacy-preserving machine learning systems requires more than identifying individual threats. It demands a layered defense strategy, which begins with protecting the data that powers models and extends through model design, deployment safeguards, runtime monitoring, and ultimately, the hardware that anchors trust. Each layer contributes to the system's overall resilience and must be tailored to the specific threat surfaces introduced by machine learning workflows. Unlike traditional software systems, ML systems are particularly vulnerable to input manipulation, data leakage, model extraction, and runtime abuse—all amplified by tight coupling between data, model behavior, and infrastructure.

This section presents a structured framework for defensive strategies, progressing from data-centric protections to infrastructure-level enforcement. These strategies span differential privacy and federated learning, robust model architectures, secure deployment pipelines, runtime validation and monitoring, and hardware-based trust anchors such as secure boot and TEEs. By integrating safeguards across layers, organizations can build ML systems that not only perform reliably but also withstand adversarial pressure in production environments.

Figure 15.12 shows a layered defense stack for machine learning systems. The stack progresses from foundational hardware-based security mechanisms to runtime system protections, model-level controls, and privacy-preserving techniques at the data level. Each layer builds on the trust guarantees of the layer below it, forming an end-to-end strategy for deploying ML systems securely. We will progressively explore each of these layers, highlighting their roles in securing machine learning systems against a range of threats.

15.7.1 Data Privacy Techniques

Protecting the privacy of individuals whose data fuels machine learning systems is a foundational requirement for trustworthy AI. Unlike traditional systems where data is often masked or anonymized before processing, ML workflows typically rely on access to raw, high-fidelity data to train effective models. This tension between utility and privacy has motivated a diverse set of techniques aimed at minimizing data exposure while preserving learning performance.

15.7.1.1 Differential Privacy

One of the most widely adopted frameworks for formalizing privacy guarantees is differential privacy (DP). DP provides a rigorous mathematical definition of privacy loss, ensuring that the inclusion or exclusion of a single individual's data has a provably limited effect on the model's output. A randomized algorithm $\mathcal{A}$ is said to be ϵ-differentially private if, for all adjacent datasets D and D'

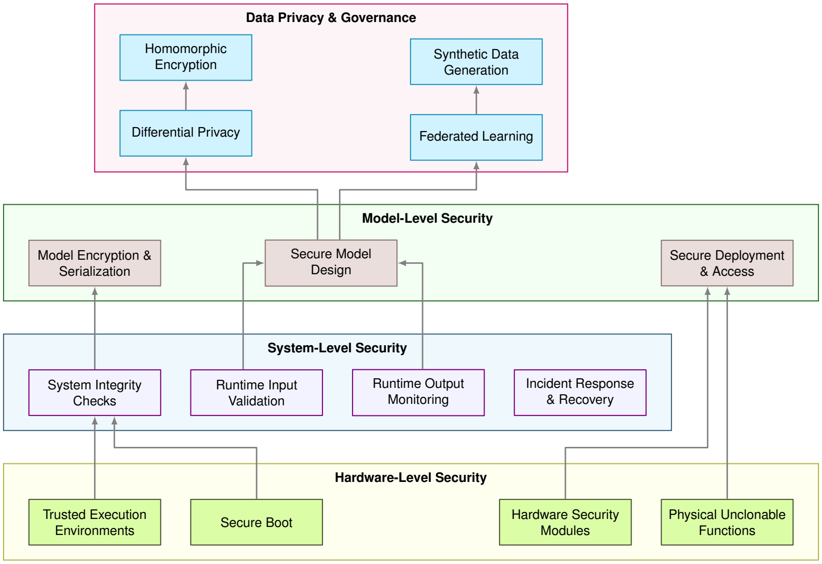

Data Privacy & Governance
Homomorphic Encryption
Synthetic Data Generation
Differential Privacy
Federated Learning
Model-Level Security
Model Encryption & Serialization
Secure Model Design
Secure Deployment & Access
System-Level Security
System Integrity Checks
Runtime Input Validation
Runtime Output Monitoring
Incident Response & Recovery
Hardware-Level Security
Trusted Execution Environments
Secure Boot
Hardware Security Modules
Physical Unclonable Functions

Figure 15.12: A layered defense stack for machine learning systems.

differing in one record, and for all outputs $S \subseteq \text{Range}(\mathcal{A})$, the following holds:

$$\Pr[\mathcal{A}(D) \in S] \leq e^{\epsilon} \Pr[\mathcal{A}(D') \in S]$$

This bound ensures that the algorithm's behavior remains statistically indistinguishable regardless of whether any individual's data is present, thereby limiting the information that can be inferred about that individual. In practice, DP is implemented by adding calibrated noise to model updates or query responses, using mechanisms such as the Laplace or Gaussian mechanism. Training techniques like differentially private stochastic gradient descent (DP-SGD) integrate noise into the optimization process to ensure per-iteration privacy guarantees.

While differential privacy offers strong theoretical assurances, it introduces a trade-off between privacy and utility. Increasing the noise to reduce ϵ may degrade model accuracy, especially in low-data regimes or fine-grained classification tasks. Consequently, DP is often applied selectively—either during training on sensitive datasets or at inference when returning aggregate statistics—to balance privacy with performance goals (Dwork and Roth 2013).

15.7.1.2 Federated Learning

Complementary to DP, federated learning (FL) reduces privacy risks by restructuring the learning process itself. Rather than aggregating raw data at a central location, FL distributes the training across a set of client devices, each holding local data (B. McMahan et al. 2017b). Clients compute model updates locally and share only parameter deltas with a central server for aggregation:

$$\theta_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} \cdot \theta_t^{(k)}$$

Here, $\theta_t^{(k)}$ represents the model update from client k, n_k the number of samples held by that client, and n the total number of samples across all clients. This weighted aggregation allows the global model to learn from distributed data without direct access to it. While FL reduces the exposure of raw data, it still leaks information through gradients, motivating the use of DP, secure aggregation, and hardware-based protections in federated settings.

To address scenarios requiring computation on encrypted data, homomorphic encryption (HE) and secure multiparty computation (SMPC) allow models to perform inference or training over encrypted inputs. In the case of HE, operations on ciphertexts correspond to operations on plaintexts, enabling encrypted inference:

$$\text{Enc}(f(x)) = f(\text{Enc}(x))$$

This property supports privacy-preserving computation in untrusted environments, such as cloud inference over sensitive health or financial records. However, the computational cost of HE remains high, making it more suitable for fixed-function models and low-latency batch tasks. SMPC, by contrast, distributes the computation across multiple parties such that no single party learns the complete input or output. This is particularly useful in joint training across institutions with strict data-use policies, such as hospitals or banks.

15.7.1.3 Synthetic Data Generation

A more pragmatic and increasingly popular alternative involves the use of synthetic data generation. By training generative models on real datasets and sampling new instances from the learned distribution, organizations can create datasets that approximate the statistical properties of the original data without retaining identifiable details (Goncalves et al. 2020). While this approach reduces the risk of direct reidentification, it does not offer formal privacy guarantees unless combined with DP constraints during generation.

Together, these techniques reflect a shift from isolating data as the sole path to privacy toward embedding privacy-preserving mechanisms into the learning process itself. Each method offers distinct guarantees and trade-offs depending on the application context, threat model, and regulatory constraints. Effective system design often combines multiple approaches, such as applying differential privacy within a federated learning setup, or employing homomorphic encryption for critical inference stages, to build ML systems that are both useful and respectful of user privacy.

15.7.1.4 Comparative Properties

These privacy-preserving techniques differ not only in the guarantees they offer but also in their system-level implications. For practitioners, the choice of mechanism depends on factors such as computational constraints, deployment architecture, and regulatory requirements.

Table 15.6 summarizes the comparative properties of these methods, focusing on privacy strength, runtime overhead, maturity, and common use cases. Understanding these trade-offs is essential for designing privacy-aware machine learning systems that operate under real-world constraints.

Table 15.6: Comparison of data privacy techniques across system-level dimensions.

Technique	Privacy Guarantee	Computational Overhead	Deployment Maturity	Typical Use Case	Trade-offs
Differential Privacy	Formal (ε-DP)	Moderate to High	Production	Training with sensitive or regulated data	Reduced accuracy; careful tuning of ε/noise required to balance utility and protection
Federated Learning	Structural	Moderate	Production	Cross-device or cross-org collaborative learning	Gradient leakage risk; requires secure aggregation and orchestration infrastructure
Homomorphic Encryption	Strong (Encrypted)	High	Experimental	Inference in untrusted cloud environments	High latency and memory usage; suitable for limited-scope inference on fixed-function models
Secure MPC	Strong (Distributed)	Very High	Experimental	Joint training across mutually untrusted parties	Expensive communication; challenging to scale to many participants or deep models
Synthetic Data	Weak (if standalone)	Low to Moderate	Emerging	Data sharing, benchmarking without direct access to raw data	May leak sensitive patterns if training process is not differentially private or audited for fidelity

15.7.2 Secure Model Design

Security begins at the design phase of a machine learning system. While downstream mechanisms such as access control and encryption protect models once deployed, many vulnerabilities can be mitigated earlier—through architectural choices, defensive training strategies, and mechanisms that embed resilience directly into the model's structure or behavior. By considering security as a design constraint, system developers can reduce the model's exposure to attacks, limit its ability to leak sensitive information, and provide verifiable ownership protection.

One important design strategy is to build robust-by-construction models that reduce the risk of exploitation at inference time. For instance, models with confidence calibration or abstention mechanisms can be trained to avoid making predictions when input uncertainty is high. These techniques can help prevent overconfident misclassifications in response to adversarial or out-of-distribution inputs. Models may also employ output smoothing, regularizing the output distribution to reduce sharp decision boundaries that are especially susceptible to adversarial perturbations.

Certain application contexts may also benefit from choosing simpler or compressed architectures. While not universally appropriate, limiting model capacity can reduce opportunities for memorization of sensitive training data and complicate efforts to reverse-engineer the model from output behavior. For embedded or on-device settings, smaller models are also easier to secure, as they typically require less memory and compute, lowering the likelihood of side-channel leakage or runtime manipulation.

Another design-stage consideration is the use of model watermarking, a technique for embedding verifiable ownership signatures directly into the model's parameters or output behavior (Adi et al. 2018). A watermark might be implemented, for example, as a hidden response pattern triggered by specific inputs, or as a parameter-space perturbation that does not affect accuracy but is statistically identifiable. These watermarks can be used to detect and prove misuse of stolen models in downstream deployments. Watermarking strategies must be carefully designed to remain robust to model compression, fine-tuning, and format conversion.

For example, in a keyword spotting system deployed on embedded hardware for voice activation (e.g., "Hey Alexa" or "OK Google"), a secure design might use a lightweight convolutional neural network with confidence calibration to avoid false activations on uncertain audio. The model might also include an abstention threshold, below which it produces no activation at all. To protect intellectual property, a designer could embed a watermark by training the model to respond with a unique label only when presented with a specific, unused audio trigger known only to the developer. These design choices not only improve robustness and accountability, but also support future verification in case of IP disputes or performance failures in the field.

In high-risk applications, such as medical diagnosis, autonomous vehicles, or financial decision systems, designers may also prioritize interpretable model architectures, such as decision trees, rule-based classifiers, or sparsified networks, to enhance system auditability. These models are often easier to understand and explain, making it simpler to identify potential vulnerabilities or biases. Using interpretable models allows developers to provide clearer insights into how the system arrived at a particular decision, which is crucial for building trust with users and regulators.

Model design choices often reflect trade-offs between accuracy, robustness, transparency, and system complexity. However, when viewed from a systems perspective, early-stage design decisions frequently yield the highest leverage for long-term security. They shape what the model can learn, how it behaves under uncertainty, and what guarantees can be made about its provenance, interpretability, and resilience.

15.7.3 Secure Model Deployment

Protecting machine learning models from theft, abuse, and unauthorized manipulation requires security considerations throughout both the design and deployment phases. A model's vulnerability is not solely determined by its training procedure or architecture, but also by how it is serialized, packaged, deployed, and accessed during inference. As models are increasingly embedded into edge devices, served through public APIs, or integrated into multi-tenant platforms, robust security practices are essential to ensure the integrity, confidentiality, and availability of model behavior.

This section addresses security mechanisms across three key stages: model design, secure packaging and serialization, and deployment and access control.

From a design perspective, architectural choices can reduce a model's exposure to adversarial manipulation and unauthorized use. For example, models can incorporate confidence calibration or abstention mechanisms that allow them to reject uncertain or anomalous inputs rather than producing potentially misleading outputs. Designing models with simpler or compressed architectures can also reduce the risk of reverse engineering or information leakage through side-channel analysis. In some cases, model designers may embed imperceptible watermarks, which are unique signatures embedded in the parameters or behavior of the model, that can later be used to demonstrate ownership in cases of misappropriation (Uchida et al. 2017). These design-time protections are particularly important for commercially valuable models, where intellectual property rights are at stake.

Once training is complete, the model must be securely packaged for deployment. Storing models in plaintext formats, including unencrypted ONNX or PyTorch checkpoint files, can expose internal structures and parameters to attackers with access to the file system or memory. To mitigate this risk, models should be encrypted, obfuscated, or wrapped in secure containers. Decryption keys should be made available only at runtime and only within trusted environments. Additional mechanisms, such as quantization-aware encryption or integrity-checking wrappers, can prevent tampering and offline model theft.

Deployment environments must also enforce strong access control policies to ensure that only authorized users and services can interact with inference endpoints. Authentication protocols, including OAuth tokens, mutual TLS, or API keys, should be combined with role-based access control (RBAC) to restrict access according to user roles and operational context. For instance, OpenAI's hosted model APIs require users to include an OPENAI_API_KEY when submitting inference requests. This key authenticates the client and enables the backend to enforce usage policies, monitor for abuse, and log access patterns. A simplified example of secure usage is shown in Listing 15.1, where the API key is securely loaded from an environment variable before being used to authenticate requests.

In this example, the API key is retrieved from an environment variable—avoiding the security risk of hardcoding it into source code or exposing it to the client side. Such key-based access control mechanisms are simple to implement but require careful key management and monitoring to prevent misuse, unauthorized access, or model extraction.

Beyond endpoint access, the integrity of the deployment pipeline itself must also be protected. Continuous integration and deployment (CI/CD) workflows that automate model updates should enforce cryptographic signing of artifacts, dependency validation, and infrastructure hardening. Without these controls, adversaries could inject malicious models or alter existing ones during the build and deployment process. Verifying model signatures and maintaining audit trails helps ensure that only authorized models are deployed into production.

When applied together, these practices protect against a range of threats—from model theft and unauthorized inference access to tampering during deployment and output manipulation at runtime. No single mechanism suffices in isolation, but a layered strategy, beginning at the design phase and extending through deployment, provides a strong foundation for securing machine learning systems under real-world conditions.

Listing 15.1: Example of securely loading an API key for OpenAI's GPT-4 model. The API key is retrieved from an environment variable to avoid hardcoding sensitive information in the source code.

```python
import openai
import os

# Securely load the API key from an environment variable
openai.api_key = os.getenv("OPENAI_API_KEY")

# Submit a prompt to the model
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[
        {
            "role": "user",
            "content": (
                "Summarize the principles of "
                "differential privacy."
            )
        }
    ]
)

print(response.choices[0].message["content"])
```

15.7.4 System-level Monitoring

Even with robust design and deployment safeguards, machine learning systems remain vulnerable to runtime threats. Attackers may craft inputs that bypass validation, exploit model behavior, or target system-level infrastructure. As ML systems enter production, particularly in cloud, edge, or embedded deployments, defensive strategies must extend beyond static protection to include real-time monitoring, threat detection, and incident response. This section outlines operational defenses that maintain system trust under adversarial conditions.

Runtime monitoring encompasses a range of techniques for observing system behavior, detecting anomalies, and triggering mitigation. These techniques can be grouped into three categories: input validation, output monitoring, and system integrity checks.

15.7.4.1 Input Validation

Input validation is the first line of defense at runtime. It ensures that incoming data conforms to expected formats, statistical properties, or semantic constraints before it is passed to a machine learning model. Without these safeguards, models are vulnerable to adversarial inputs, which are crafted examples designed to trigger incorrect predictions, or to malformed inputs that cause unexpected behavior in preprocessing or inference.

Machine learning models, unlike traditional rule-based systems, often do not fail safely. Small, carefully chosen changes to input data can cause models to make high-confidence but incorrect predictions. Input validation helps detect and reject such inputs early in the pipeline (I. J. Goodfellow, Shlens, and Szegedy 2014).

Validation techniques range from low-level checks (e.g., input size, type, and value ranges) to semantic filters (e.g., verifying whether an image contains a recognizable object or whether a voice recording includes speech). For example, a facial recognition system might validate that the uploaded image is within a certain resolution range (e.g., 224×224 to 1024×1024 pixels), contains RGB channels, and passes a lightweight face detection filter. This prevents inputs like blank images, text screenshots, or synthetic adversarial patterns from reaching the model. Similarly, a voice assistant might require that incoming audio files be between 1 and 5 seconds long, have a valid sampling rate (e.g., 16kHz), and contain detectable human speech using a speech activity detector (SAD). This ensures that empty recordings, music clips, or noise bursts are filtered before model inference.

In generative systems such as DALL·E, Stable Diffusion, or Sora, input validation often involves prompt filtering. This includes scanning the user's text prompt for banned terms, brand names, profanity, or misleading medical claims. For example, a user prompt like "Generate an image of a medication bottle labeled with Pfizer's logo" might be rejected or rewritten due to trademark concerns. Filters may operate using keyword lists, regular expressions, or lightweight classifiers that assess prompt intent. These filters prevent the generative model from being used to produce harmful, illegal, or misleading content—even before sampling begins.

In some applications, distributional checks are also used. These assess whether the incoming data statistically resembles what the model saw during training. For instance, a computer vision pipeline might compare the color histogram of the input image to a baseline distribution, flagging outliers for manual review or rejection.

These validations can be lightweight (heuristics or threshold rules) or learned (small models trained to detect distribution shift or adversarial artifacts). In either case, input validation serves as a critical pre-inference firewall—reducing exposure to adversarial behavior, improving system stability, and increasing trust in downstream model decisions.

15.7.4.2 Output Monitoring

Even when inputs pass validation, adversarial or unexpected behavior may still emerge at the model's output. Output monitoring helps detect such anomalies by analyzing model predictions in real time. These mechanisms observe how the model behaves across inputs, by tracking its confidence, prediction entropy, class distribution, or response patterns, to flag deviations from expected behavior.

A key target for monitoring is prediction confidence. For example, if a classification model begins assigning high confidence to low-frequency or previously rare classes, this may indicate the presence of adversarial inputs or a shift in the underlying data distribution. Monitoring the entropy of the output distribution can similarly reveal when the model is overly certain in ambiguous contexts—an early signal of possible manipulation.

In content moderation systems, a model that normally outputs neutral or "safe" labels may suddenly begin producing high-confidence "safe" labels for inputs containing offensive or restricted content. Output monitoring can detect this mismatch by comparing predictions against auxiliary signals or known-safe reference sets. When deviations are detected, the system may trigger a fallback policy—such as escalating the content for human review or switching to a conservative baseline model.

Time-series models also benefit from output monitoring. For instance, an anomaly detection model used in fraud detection might track predicted fraud scores for sequences of financial transactions. A sudden drop in fraud scores, especially during periods of high transaction volume, may indicate model tampering, label leakage, or evasion attempts. Monitoring the temporal evolution of predictions provides a broader perspective than static, pointwise classification.

Generative models, such as text-to-image systems, introduce unique output monitoring challenges. These models can produce high-fidelity imagery that may inadvertently violate content safety policies, platform guidelines, or user expectations. To mitigate these risks, post-generation classifiers are commonly employed to assess generated content for objectionable characteristics such as violence, nudity, or brand misuse. These classifiers operate downstream of the generative model and can suppress, blur, or reject outputs based on predefined thresholds. Some systems also inspect internal representations (e.g., attention maps or latent embeddings) to anticipate potential misuse before content is rendered.

However, prompt filtering alone is insufficient for safety. Research has shown that text-to-image systems can be manipulated through implicitly adversarial prompts, which are queries that appear benign but lead to policy-violating outputs. The Adversarial Nibbler project introduces an open red teaming methodology that identifies such prompts and demonstrates how models like Stable Diffusion can produce unintended content despite the absence of explicit trigger phrases (Quaye et al. 2024). These failure cases often bypass prompt filters because their risk arises from model behavior during generation, not from syntactic or lexical cues.

Figure 15.13: Example of an implicitly adversarial prompt ("splatter of red paint") generating unintended content in a text-to-image system. These types of failures bypass prompt filters and highlight the need for post-generation safety monitoring. Source: Adapted from Quaye et al. (2024).

As shown in Figure 15.13, even prompts that appear innocuous can trigger unsafe generations. Such examples highlight the limitations of pre-generation safety checks and reinforce the necessity of output-based monitoring as a second line of defense. This two-stage pipeline—consisting of prompt filtering followed by post-hoc content analysisis essential for ensuring the safe deployment of generative models in open-ended or user-facing environments.

In the domain of language generation, output monitoring plays a different but equally important role. Here, the goal is often to detect toxicity, hallucinated claims, or off-distribution responses. For example, a customer support chatbot may be monitored for keyword presence, tonal alignment, or semantic coherence. If a response contains profanity, unsupported assertions, or syntactically malformed text, the system may trigger a rephrasing, initiate a fallback to scripted templates, or halt the response altogether.

Effective output monitoring combines rule-based heuristics with learned detectors trained on historical outputs. These detectors are deployed to flag deviations in real time and feed alerts into incident response pipelines. In contrast to model-centric defenses like adversarial training, which aim to improve model robustness, output monitoring emphasizes containment and remediation. Its role is not to prevent exploitation but to detect its symptoms and initiate appropriate countermeasures (Savas et al. 2022). In safety-critical or policy-sensitive applications, such mechanisms form a critical layer of operational resilience.

These principles have been implemented in recent output filtering frameworks. For example, LLM Guard combines transformer-based classifiers with safety dimensions such as toxicity, misinformation, and illegal content to assess and reject prompts or completions in instruction-tuned LLMs (Inan et al. 2023). Similarly, ShieldGemma, developed as part of Google's open Gemma model release, applies configurable scoring functions to detect and filter undesired outputs during inference[7]. Both systems exemplify how safety classifiers and output monitors are being integrated into the runtime stack to support scalable, policy-aligned deployment of generative language models.

15.7.4.3 Integrity Checks

While input and output monitoring focus on model behavior, system integrity checks ensure that the underlying model files, execution environment, and serving infrastructure remain untampered throughout deployment. These checks detect unauthorized modifications, verify that the model running in production is authentic, and alert operators to suspicious system-level activity.

One of the most common integrity mechanisms is cryptographic model verification. Before a model is loaded into memory, the system can compute a cryptographic hash (e.g., SHA-256) of the model file and compare it against a known-good signature. This process ensures that the model has not been altered during transit or storage. For example, a PyTorch .pt or TensorFlow .pb model artifact stored in object storage (e.g., S3) might be verified using a signed hash from a deployment registry before loading into a production container. If the verification fails, the system can block inference, alert an operator, or revert to a trusted model version.

Access control and audit logging complement cryptographic checks. ML systems should restrict access to model files using role-based permissions and monitor file access patterns. For instance, repeated attempts to read model checkpoints from a non-standard path, or inference requests from unauthorized IP ranges, may indicate tampering, privilege escalation, or insider threats.

[7] ShieldGemma is a framework for filtering outputs from large language models, developed by Google as part of the Gemma model release. It applies configurable scoring functions to detect and filter undesired outputs during inference.

In cloud environments, container- or VM-based isolation helps enforce process and memory boundaries, but these protections can erode over time due to misconfiguration or supply chain vulnerabilities. To reinforce runtime assurance, systems may deploy periodic attestation checks—verifying not just the model artifact, but also the software environment, installed dependencies, and hardware identity. These techniques are often backed by hardware trust anchors (e.g., TPMs or TEEs) discussed later on in this chapter.

For example, in a regulated healthcare ML deployment, integrity checks might include: verifying the model hash against a signed manifest, validating that the runtime environment uses only approved Python packages, and checking that inference occurs inside a signed and attested virtual machine. These checks ensure compliance, limit the risk of silent failures, and create a forensic trail in case of audit or breach.

Some systems also implement runtime memory verification, such as scanning for unexpected model parameter changes or checking that memory-mapped model weights remain unaltered during execution. While more common in high-assurance systems, such checks are becoming more feasible with the adoption of secure enclaves and trusted runtimes.

Taken together, system integrity checks play a critical role in protecting machine learning systems from low-level attacks that bypass the model interface. When coupled with input/output monitoring, they provide layered assurance that both the model and its execution environment remain trustworthy under adversarial conditions.

15.7.4.4 Response and Rollback

When a security breach, anomaly, or performance degradation is detected in a deployed machine learning system, rapid and structured incident response is critical to minimizing impact. The goal is not only to contain the issue but to restore system integrity and ensure that future deployments benefit from the insights gained. Unlike traditional software systems, ML responses may require handling model state, data drift, or inference behavior, making recovery more complex.

The first step is to define incident detection thresholds that trigger escalation. These thresholds may come from input validation (e.g., invalid input rates), output monitoring (e.g., drop in prediction confidence), or system integrity checks (e.g., failed model signature verification). When a threshold is crossed, the system should initiate an automated or semi-automated response protocol.

One common strategy is model rollback, where the system reverts to a previously verified version of the model. For instance, if a newly deployed fraud detection model begins misclassifying transactions, the system may fall back to the last known-good checkpoint, restoring service while the affected version is quarantined. Rollback mechanisms require version-controlled model storage, typically supported by MLOps platforms such as MLflow, TFX, or SageMaker.

In high-availability environments, model isolation may be used to contain failures. The affected model instance can be removed from load balancers or shadowed in a canary deployment setup. This allows continued service with unaffected replicas while maintaining forensic access to the compromised model for analysis.

Traffic throttling is another immediate response tool. If an adversarial actor is probing a public inference API at high volume, the system can rate-limit or temporarily block offending IP ranges while continuing to serve trusted clients. This containment technique helps prevent abuse without requiring full system shutdown.

Once immediate containment is in place, investigation and recovery can begin. This may include forensic analysis of input logs, parameter deltas between model versions, or memory snapshots from inference containers. In regulated environments, organizations may also need to notify users or auditors, particularly if personal or safety-critical data was affected.

Recovery typically involves retraining or patching the model. This must occur through a secure update process, using signed artifacts, trusted build pipelines, and validated data. To prevent recurrence, the incident should feed back into model evaluation pipelines—updating tests, refining monitoring thresholds, or hardening input defenses. For example, if a prompt injection attack bypassed a content filter in a generative model, retraining might include adversarially crafted prompts, and the prompt validation logic would be updated to reflect newly discovered patterns.

Finally, organizations should establish post-incident review practices. This includes documenting root causes, identifying gaps in detection or response, and updating policies and playbooks. Incident reviews help translate operational failures into actionable improvements across the design-deploy-monitor lifecycle.

15.7.5 Hardware-based Security

Machine learning systems are increasingly deployed in environments where hardware-based security features can provide additional layers of protection. These features can help ensure the integrity of model execution, protect sensitive data, and prevent unauthorized access to system resources. This section discusses several key hardware-based security mechanisms that can enhance the security posture of machine learning systems.

15.7.5.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a hardware-isolated region within a processor designed to protect sensitive computations and data from potentially compromised software. TEEs enforce confidentiality, integrity, and runtime isolation, ensuring that even if the host operating system or application layer is attacked, sensitive operations within the TEE remain secure.

In the context of machine learning, TEEs are increasingly important for preserving the confidentiality of models, securing sensitive user data during inference, and ensuring that model outputs remain trustworthy. For example, a TEE can protect model parameters from being extracted by malicious software running on the same device, or ensure that computations involving biometric inputs, including facial data or fingerprint data, are performed securely. This capability is particularly critical in applications where model integrity, user privacy, or regulatory compliance are non-negotiable.

One widely deployed example is Apple's Secure Enclave, which provides isolated execution and secure key storage for iOS devices. By separating cryptographic operations and biometric data from the main processor, the Secure Enclave ensures that user credentials and Face ID features remain protected, even in the event of a broader system compromise.

Trusted Execution Environments are essential across a range of industries with high security requirements. In telecommunications, TEEs are used to safeguard encryption keys and secure critical 5G control-plane operations. In finance, they enable secure mobile payments and protect PIN-based authentication workflows. In healthcare, TEEs help enforce patient data confidentiality during edge-based ML inference on wearable or diagnostic devices. In the automotive industry, they are deployed in advanced driver-assistance systems (ADAS) to ensure that safety-critical perception and decision-making modules operate on verified software.

In machine learning systems, TEEs can provide several important protections. They secure the execution of model inference or training, shielding intermediate computations and final predictions from system-level observation. They protect the confidentiality of sensitive inputs, including biometric or clinical signals, used in personal identification or risk scoring tasks. TEEs also serve to prevent reverse engineering of deployed models by restricting access to weights and architecture internals. When models are updated, TEEs ensure the authenticity of new parameters and block unauthorized tampering. Furthermore, in distributed ML settings, TEEs can protect data exchanged between components by enabling encrypted and attested communication channels.

The core security properties of a TEE are achieved through four mechanisms: isolated execution, secure storage, integrity protection, and in-TEE data encryption. Code that runs inside the TEE is executed in a separate processor mode, inaccessible to the normal-world operating system. Sensitive assets such as cryptographic keys or authentication tokens are stored in memory that only the TEE can access. Code and data can be verified for integrity before execution using hardware-anchored hashes or signatures. Finally, data processed inside the TEE can be encrypted, ensuring that even intermediate results are inaccessible without appropriate keys, which are also managed internally by the TEE.

Several commercial platforms provide TEE functionality tailored for different deployment contexts. ARM TrustZone offers secure and normal world execution on ARM-based systems and is widely used in mobile and IoT applications. Intel SGX implements enclave-based security for cloud and desktop systems, enabling secure computation even on untrusted infrastructure. Qualcomm's Secure Execution Environment supports secure mobile transactions and user authentication. Apple's Secure Enclave remains a canonical example of a hardware-isolated security coprocessor for consumer devices.

Figure 15.14 illustrates a secure enclave integrated into a system-on-chip (SoC) architecture. The enclave includes a dedicated processor, an AES engine, a true random number generator (TRNG), a public key accelerator (PKA), and a secure I^{2}C interface to nonvolatile storage. These components operate in isolation from the main application processor and memory subsystem. A memory protection engine enforces access control, while cryptographic operations such as NAND flash encryption are handled internally using enclave-managed keys.

By physically separating secure execution and key management from the main system, this architecture limits the impact of system-level compromises and forms the foundation of hardware-enforced trust.

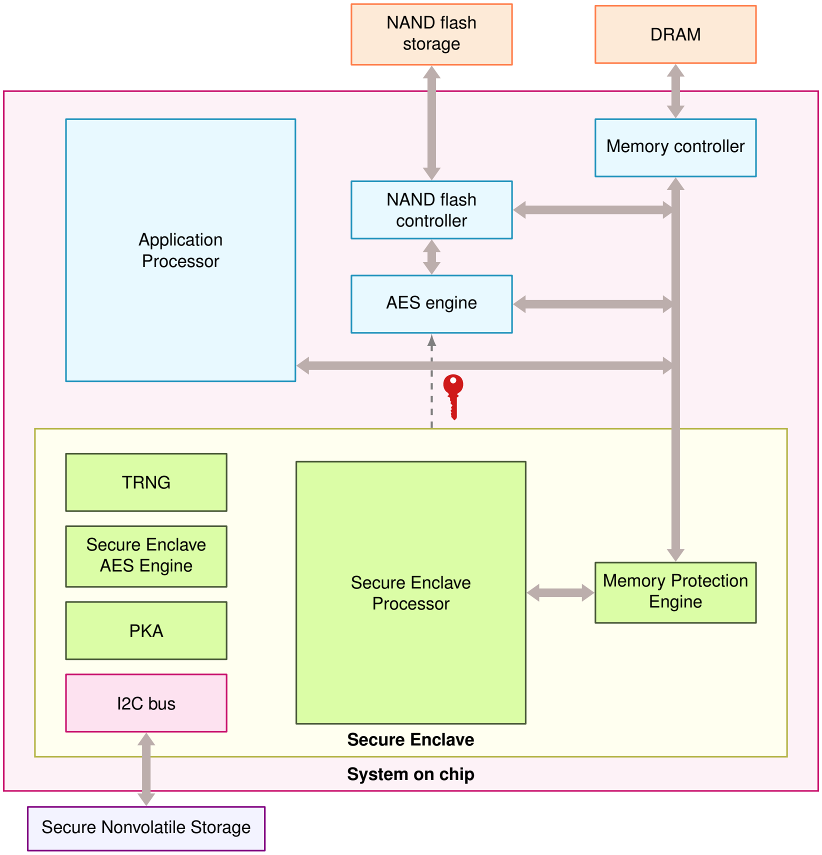


Figure 15.14: System-on-chip secure enclave. Source: Apple.

This architecture underpins the secure deployment of machine learning applications on consumer devices. For example, Apple's Face ID system uses a secure enclave to perform facial recognition entirely within a hardware-isolated environment. The face embedding model is executed inside the enclave, and biometric templates are stored in secure nonvolatile memory accessible only via the enclave's I^{2}C interface. During authentication, input data from the infrared camera is processed locally, and no facial features or predictions ever leave the secure region. Even if the application processor or operating system is compromised, the enclave prevents access to sensitive model inputs, parameters, and outputs—ensuring that biometric identity remains protected end to end.

Despite their strengths, Trusted Execution Environments come with notable trade-offs. Implementing a TEE increases both direct hardware costs and indirect costs associated with developing and maintaining secure software. Integrating TEEs into existing systems may require architectural redesigns, especially for legacy infrastructure. Developers must adhere to strict protocols for isolation, attestation, and secure update management, which can extend development cycles and complicate testing workflows. TEEs can also introduce performance overhead, particularly when cryptographic operations are involved, or when context switching between trusted and untrusted modes is frequent.

Energy efficiency is another consideration, particularly in battery-constrained devices. TEEs typically consume additional power due to secure memory accesses, cryptographic computation, and hardware protection logic. In resource-limited embedded systems, these costs may limit their use. In terms of scalability and flexibility, the secure boundaries enforced by TEEs may complicate distributed training or federated inference workloads, where secure coordination between enclaves is required.

Market demand also varies. In some consumer applications, perceived threat levels may be too low to justify the integration of TEEs. Moreover, systems with TEEs may be subject to formal security certifications, such as Common Criteria or evaluation under ENISA, which can introduce additional time and expense. For this reason, TEEs are typically adopted only when the expected threat model, including adversarial users, cloud tenants, and malicious insiders, justifies the investment.

Nonetheless, TEEs remain a powerful hardware primitive in the machine learning security landscape. When paired with software- and system-level defenses, they provide a trusted foundation for executing ML models securely, privately, and verifiably, especially in scenarios where adversarial compromise of the host environment is a serious concern.

Here is the revised 7.5.2 Secure Boot section, rewritten in formal textbook tone with all original technical content, hyperlinks, and figures preserved. The structure emphasizes narrative clarity, avoids bullet lists, and integrates the Apple Face ID case study naturally.

15.7.5.2 Secure Boot

Secure Boot is a mechanism that ensures a device only boots software components that are cryptographically verified and explicitly authorized by the manufacturer. At startup, each stage of the boot process, comprising the bootloader, kernel, and base operating system, is checked against a known-good digital signature. If any signature fails verification, the boot sequence is halted, preventing unauthorized or malicious code from executing. This chain-of-trust model establishes system integrity from the very first instruction executed.

In ML systems, especially those deployed on embedded or edge hardware, Secure Boot plays an important role. A compromised boot process may result in malicious software loading before the ML runtime begins, enabling attackers to intercept model weights, tamper with training data, or reroute inference results. Such breaches can lead to incorrect or manipulated predictions, unauthorized data access, or device repurposing for botnets or crypto-mining.

For machine learning systems, Secure Boot offers several guarantees. First, it protects model-related data, such as training data, inference inputs, and outputs, during the boot sequence, preventing pre-runtime tampering. Second, it ensures that only authenticated model binaries and supporting software are loaded, which helps guard against deployment-time model substitution. Third, Secure Boot enables secure model updates by verifying that firmware or model changes are signed and have not been altered in transit.

Secure Boot frequently works in tandem with hardware-based Trusted Execution Environments (TEEs) to create a fully trusted execution stack. As shown in Figure 15.15, this layered boot process verifies firmware, operating system components, and TEE integrity before permitting execution of cryptographic operations or ML workloads. In embedded systems, this architecture provides resilience even under severe adversarial conditions or physical device compromise.

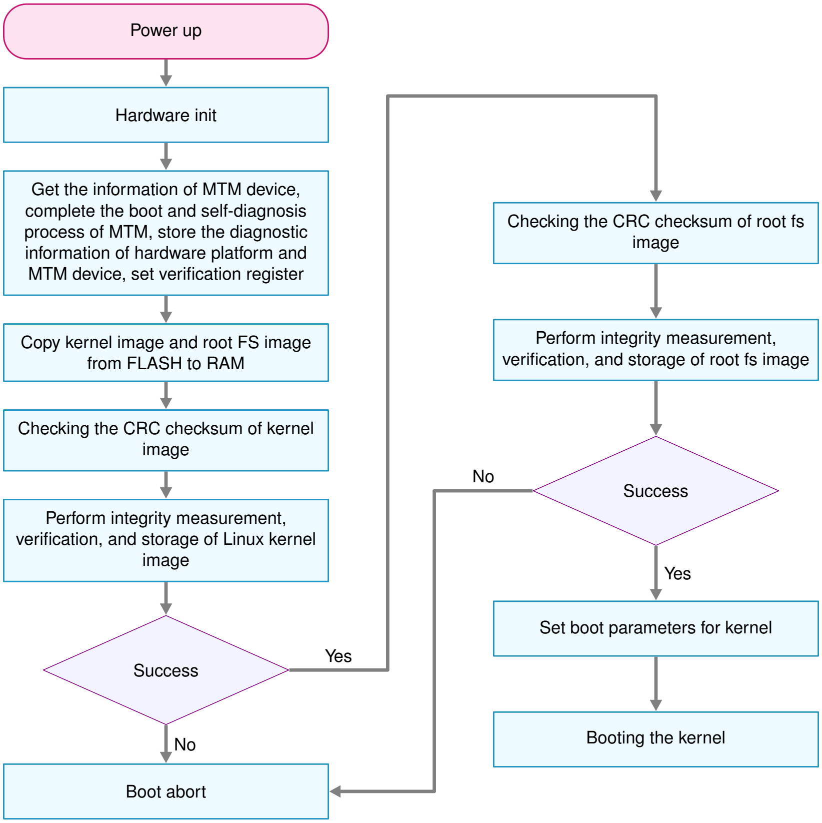

Power up
Hardware init
Get the information of MTM device, complete the boot and self-diagnosis process of MTM, store the diagnostic information of hardware platform and MTM device, set verification register
Copy kernel image and root FS image from FLASH to RAM
Checking the CRC checksum of kernel image
Perform integrity measurement, verification, and storage of Linux kernel image
Success
Yes
No
Boot abort
Checking the CRC checksum of root fs image
Perform integrity measurement, verification, and storage of root fs image
Success
No
Yes
Set boot parameters for kernel
Booting the kernel

Figure 15.15: Secure Boot flow. Source: R. V. and A. (2018).

A well-known real-world implementation of Secure Boot appears in Apple's Face ID system, which leverages advanced machine learning for facial recognition. For Face ID to operate securely, the entire device stack, from the initial power-on to the execution of the model, must be verifiably trusted.

Upon device startup, Secure Boot initiates within Apple's Secure Enclave, a dedicated security coprocessor that handles biometric data. The firmware loaded onto the Secure Enclave is digitally signed by Apple, and any unauthorized modification causes the boot process to fail. Once verified, the Secure

Enclave performs continuous checks in coordination with the central processor to maintain a trusted boot chain. Each system component, ranging from the iOS kernel to the application-level code, is verified using cryptographic signatures.

After completing the secure boot sequence, the Secure Enclave activates the ML-based Face ID system. The facial recognition model projects over 30,000 infrared points to map a user's face, generating a depth image and computing a mathematical representation that is compared against a securely stored profile. These facial data artifacts are never written to disk, transmitted off-device, or shared externally. All processing occurs within the enclave to protect against eavesdropping or exfiltration, even in the presence of a compromised kernel.

To support continued integrity, Secure Boot also governs software updates. Only firmware or model updates signed by Apple are accepted, ensuring that even over-the-air patches do not introduce risk. This process maintains a robust chain of trust over time, enabling the secure evolution of the ML system while preserving user privacy and device security.

While Secure Boot provides strong protection, its adoption presents technical and operational challenges. Managing the cryptographic keys used to sign and verify system components is complex, especially at scale. Enterprises must securely provision, rotate, and revoke keys, ensuring that no trusted root is compromised. Any such breach would undermine the entire security chain.

Performance is also a consideration. Verifying signatures during the boot process introduces latency, typically on the order of tens to hundreds of milliseconds per component. Although acceptable in many applications, these delays may be problematic for real-time or power-constrained systems. Developers must also ensure that all components, including bootloaders, firmware, kernels, drivers, and even ML models, are correctly signed. Integrating third-party software into a Secure Boot pipeline introduces additional complexity.

Some systems limit user control in favor of vendor-locked security models, restricting upgradability or customization. In response, open-source bootloaders like u-boot and coreboot have emerged, offering Secure Boot features while supporting extensibility and transparency. To further scale trusted device deployments, emerging industry standards such as the Device Identifier Composition Engine (DICE) and IEEE 802.1AR IDevID provide mechanisms for secure device identity, key provisioning, and cross-vendor trust assurance.

Secure Boot, when implemented carefully and complemented by trusted hardware and secure software update processes, forms the backbone of system integrity for embedded and distributed ML. It provides the assurance that the machine learning model running in production is not only the correct version, but is also executing in a known-good environment, anchored to hardware-level trust.

15.7.5.3 Hardware Security Modules

A Hardware Security Module (HSM) is a tamper-resistant physical device designed to perform cryptographic operations and securely manage digital keys. HSMs are widely used across security-critical industries such as finance, defense, and cloud infrastructure, and they are increasingly relevant for securing the machine learning pipeline—particularly in deployments where key confidentiality, model integrity, and regulatory compliance are essential.

HSMs provide an isolated, hardened environment for performing sensitive operations such as key generation, digital signing, encryption, and decryption. Unlike general-purpose processors, they are engineered to withstand physical tampering and side-channel attacks, and they typically include protected storage, cryptographic accelerators, and internal audit logging. HSMs may be implemented as standalone appliances, plug-in modules, or integrated chips embedded within broader systems.

In machine learning systems, HSMs enhance security across several dimensions. They are commonly used to protect encryption keys associated with sensitive data that may be processed during training or inference. These keys might encrypt data at rest in model checkpoints or enable secure transmission of inference requests across networked environments. By ensuring that the keys are generated, stored, and used exclusively within the HSM, the system minimizes the risk of key leakage, unauthorized reuse, or tampering.

HSMs also play a role in maintaining the integrity of machine learning models. In many production pipelines, models must be signed before deployment to ensure that only verified versions are accepted into runtime environments. The signing keys used to authenticate models can be stored and managed within the HSM, providing cryptographic assurance that the deployed artifact is authentic and untampered. Similarly, secure firmware updates and configuration changes, regardless of whether they pertain to models, hyperparameters, or supporting infrastructure, can be validated using signatures produced by the HSM.

In addition to protecting inference workloads, HSMs can be used to secure model training. During training, data may originate from distributed and potentially untrusted sources. HSM-backed protocols can help ensure that training pipelines perform encryption, integrity checks, and access control enforcement securely and in compliance with organizational or legal requirements. In regulated industries such as healthcare and finance, such protections are often mandatory.

Despite these benefits, incorporating HSMs into embedded or resource-constrained ML systems introduces several trade-offs. First, HSMs are specialized hardware components and often come at a premium. Their cost may be justified in data center settings or safety-critical applications but can be prohibitive for low-margin embedded products or wearables. Physical space is also a concern. Embedded systems often operate under strict size, weight, and form factor constraints, and integrating an HSM may require redesigning circuit layouts or sacrificing other functionality.

From a performance standpoint, HSMs introduce latency, particularly for operations like key exchange, signature verification, or on-the-fly decryption. In real-time inference systems, including autonomous vehicles, industrial robotics, and live translation devices, these delays can affect responsiveness. While HSMs are typically optimized for cryptographic throughput, they are not general-purpose processors, and offloading secure operations must be carefully coordinated.

Power consumption is another concern. The continuous secure handling of keys, signing of transactions, and cryptographic validations can consume more power than basic embedded components, impacting battery life in mobile or remote deployments.

Integration complexity also grows when HSMs are introduced into existing ML pipelines. Interfacing between the HSM and the host processor requires dedicated APIs and often specialized software development. Firmware and model updates must be routed through secure, signed channels, and update orchestration must account for device-specific key provisioning. These requirements increase the operational burden, especially in large deployments.

Scalability presents its own set of challenges. Managing a distributed fleet of HSM-equipped devices requires secure provisioning of individual keys, secure identity binding, and coordinated trust management. In large ML deployments, including fleets of smart sensors or edge inference nodes, ensuring uniform security posture across all devices is nontrivial.

Finally, the use of HSMs often requires organizations to engage in certification and compliance processes, particularly when handling regulated data. Meeting standards such as FIPS 140-2 or Common Criteria adds time and cost to development. Access to the HSM is typically restricted to a small set of authorized personnel, which can complicate development workflows and slow iteration cycles.

Despite these operational complexities, HSMs remain a valuable option for machine learning systems that require high assurance of cryptographic integrity and access control. When paired with TEEs, secure boot, and software-based defenses, HSMs contribute to a multilayered security model that spans hardware, system software, and ML runtime.

15.7.5.4 Physical Unclonable Functions

Physical Unclonable Functions (PUFs) provide a hardware-intrinsic mechanism for cryptographic key generation and device authentication by leveraging physical randomness in semiconductor fabrication (Gassend et al. 2002). Unlike traditional keys stored in memory, a PUF generates secret values based on microscopic variations in a chip's physical properties—variations that are inherent to manufacturing processes and difficult to clone or predict, even by the manufacturer.

These variations arise from uncontrollable physical factors such as doping concentration, line edge roughness, and dielectric thickness. As a result, even chips fabricated with the same design masks exhibit small but measurable differences in timing, power consumption, or voltage behavior. PUF circuits amplify these variations to produce a device-unique digital output. When a specific input challenge is applied to a PUF, it generates a corresponding response based on the chip's physical fingerprint. Because these characteristics are effectively impossible to replicate, the same challenge will yield different responses across devices.

This challenge-response mechanism allows PUFs to serve several cryptographic purposes. They can be used to derive device-specific keys that never need to be stored externally, reducing the attack surface for key exfiltration. The same mechanism also supports secure authentication and attestation, where devices must prove their identity to trusted servers or hardware gateways. These properties make PUFs a natural fit for machine learning systems deployed in embedded and distributed environments.

In ML applications, PUFs offer unique advantages for securing resource-constrained systems. For example, consider a smart camera drone that uses onboard computer vision to track objects. A PUF embedded in the drone's processor can generate a private key to encrypt the model during boot. Even if the model were extracted, it would be unusable on another device lacking the same PUF response. That same PUF-derived key could also be used to watermark the model parameters, creating a cryptographically verifiable link between a deployed model and its origin hardware. If the model were leaked or pirated, the embedded watermark could help prove the source of the compromise.

PUFs also support authentication in distributed ML pipelines. If the drone offloads computation to a cloud server, the PUF can help verify that the drone has not been cloned or tampered with. The cloud backend can issue a challenge, verify the correct response from the device, and permit access only if the PUF proves device authenticity. These protections enhance trust not only in the model and data, but in the execution environment itself.

The internal operation of a PUF is illustrated in Figure 15.16. At a high level, a PUF accepts a challenge input and produces a unique response determined by the physical microstructure of the chip (Gao, Al-Sarawi, and Abbott 2020). Variants include optical PUFs, in which the challenge consists of a light pattern and the response is a speckle image, and electronic PUFs such as Arbiter PUFs (APUFs), where timing differences between circuit paths produce a binary output. Another common implementation is the SRAM PUF, which exploits the power-up state of uninitialized SRAM cells: due to threshold voltage mismatch, each cell tends to settle into a preferred value when power is first applied. These response patterns form a stable, reproducible hardware fingerprint.

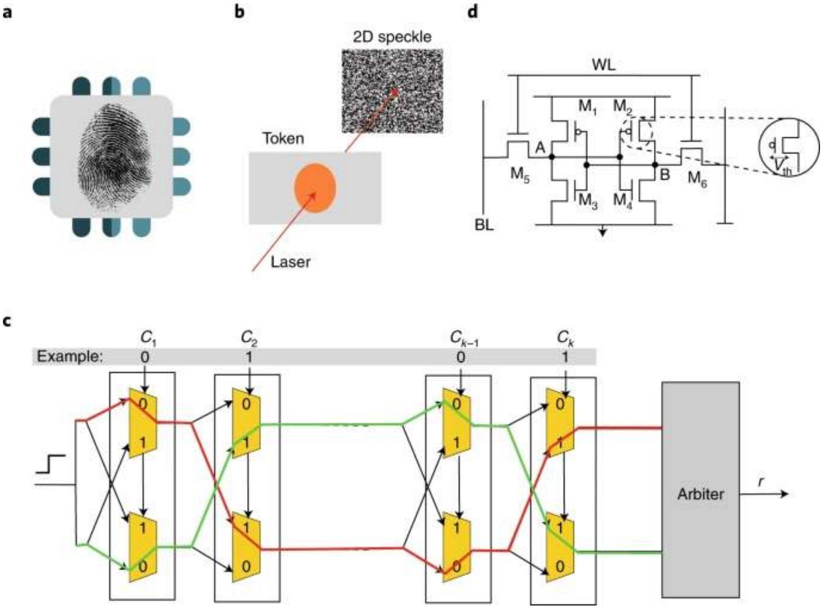


Figure 15.16: PUF basics. Source: Gao, Al-Sarawi, and Abbott (2020).

Despite their promise, PUFs present several challenges in system design. Their outputs can be sensitive to environmental variation, such as changes in temperature or voltage, which can introduce instability or bit errors in the response. To ensure reliability, PUF systems must often incorporate error correction codes or helper data schemes. Managing large sets of challenge-response pairs also raises questions about storage, consistency, and revocation. Additionally, the unique statistical structure of PUF outputs may make them vulnerable to machine learning-based modeling attacks if not carefully shielded from external observation.

From a manufacturing perspective, incorporating PUF technology can increase device cost or require additional layout complexity. While PUFs eliminate the need for external key storage, thereby reducing long-term security risk and provisioning cost, they may require calibration and testing during fabrication to ensure consistent performance across environmental conditions and device aging.

Nevertheless, Physical Unclonable Functions remain a compelling building block for securing embedded machine learning systems. By embedding hardware identity directly into the chip, PUFs support lightweight cryptographic operations, reduce key management burden, and help establish root-of-trust anchors in distributed or resource-constrained environments. When integrated thoughtfully, they complement other hardware-assisted security mechanisms such as Secure Boot, TEEs, and HSMs to provide defense-in-depth across the ML system lifecycle.

15.7.5.5 Mechanisms Comparison

Hardware-assisted security mechanisms play a foundational role in establishing trust within modern machine learning systems. While software-based defenses offer flexibility, they ultimately rely on the security of the hardware platform. As machine learning workloads increasingly operate on edge devices, embedded platforms, and untrusted infrastructure, hardware-backed protections become essential for maintaining system integrity, confidentiality, and trust.

Trusted Execution Environments (TEEs) provide runtime isolation for model inference and sensitive data handling. Secure Boot enforces integrity from power-on, ensuring that only verified software is executed. Hardware Security Modules (HSMs) offer tamper-resistant storage and cryptographic processing for secure key management, model signing, and firmware validation. Physical Unclonable Functions (PUFs) bind secrets and authentication to the physical characteristics of a specific device, enabling lightweight and unclonable identities.

These mechanisms address different layers of the system stack, ranging from initialization and attestation to runtime protection and identity binding, and complement one another when deployed together. Table 15.7 below compares their roles, use cases, and trade-offs in machine learning system design.

Table 15.7: Hardware security mechanisms comparison.

Mechanism	Primary Function	Common Use in ML	Trade-offs
Trusted Execution Environment (TEE)	Isolated runtime environment for secure computation	Secure inference and on-device privacy for sensitive inputs and outputs	Added complexity, memory limits, perf. cost Requires trusted code development
Secure Boot	Verified boot sequence and firmware validation	Ensures only signed ML models and firmware execute on embedded devices	Key management complexity, vendor lock-in Performance impact during startup
Hardware Security Module (HSM)	Secure key generation and storage, crypto-processing	Signing ML models, securing training pipelines, verifying firmware	High cost, integration overhead, limited I/O Requires infrastructure-level provisioning
Physical Unclonable Function (PUF)	Hardware-bound identity and key derivation	Model binding, device authentication, protecting IP in embedded deployments	Environmental sensitivity, modeling attacks Needs error correction and calibration

Together, these hardware primitives form the foundation of a defense-in-depth strategy for securing ML systems in adversarial environments. Their integration is especially important in domains that demand provable trust, such as autonomous vehicles, healthcare devices, federated learning systems, and critical infrastructure.

15.7.6 Toward Trustworthy Systems

Defending machine learning systems against adversarial threats, misuse, and system compromise requires more than isolated countermeasures. As this section has shown, effective defense emerges from the careful integration of mechanisms at multiple layers of the ML stack—from privacy-preserving data handling and robust model design to runtime monitoring and hardware-enforced isolation. No single component can provide complete protection; instead, a trustworthy system is the result of coordinated design decisions that address risk across the data, model, system, and infrastructure layers.

Defensive strategies must align with the deployment context and threat model. What is appropriate for a public cloud API may differ from the requirements of an embedded medical device or a fleet of edge-deployed sensors. Design choices must balance security, performance, and usability, recognizing that protections often introduce operational trade-offs. Monitoring and incident response mechanisms ensure resilience during live operation, while hardware-based roots of trust ensure system integrity even when higher layers are compromised.

As machine learning continues to expand into safety-critical, privacy-sensitive, and decentralized environments, the need for robust, end-to-end defense becomes increasingly urgent. Building ML systems that are not only accurate, but secure, private, and auditable, is fundamental to long-term deployment success and public trust. The principles introduced in this section lay the groundwork for such systems—while connecting forward to broader concerns explored in subsequent chapters, including robustness, responsible AI, and MLOps operations.

The process of engineering trustworthy ML systems requires a structured approach that connects threat modeling to layered defenses and runtime resilience.

Figure 15.17 provides a conceptual framework to guide this process across technical and deployment dimensions. The design flow begins with a thorough assessment of the threat model and deployment context, which informs the selection of appropriate defenses across the system stack. This includes data-layer protections such as differential privacy (DP), federated learning (FL), and encryption; model-layer defenses like robustness techniques, watermarking, and secure deployment practices; runtime-layer measures such as input validation and output monitoring; and hardware-layer solutions including TEEs, secure boot, and PUFs.

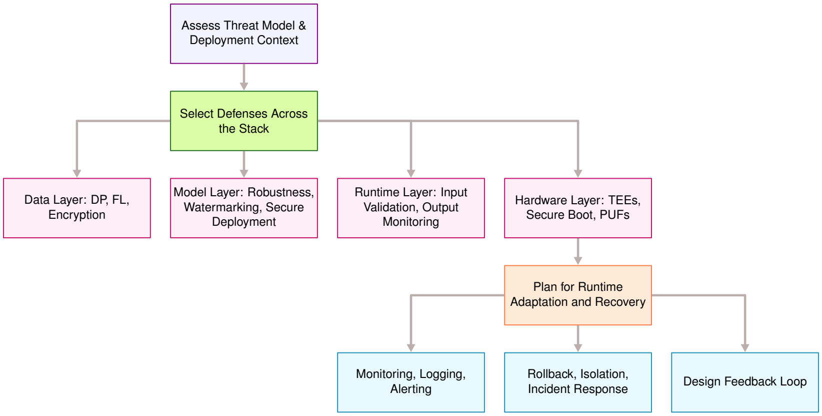

Assess Threat Model & Deployment Context
Select Defenses Across the Stack
Data Layer: DP, FL, Encryption
Model Layer: Robustness, Watermarking, Secure Deployment
Runtime Layer: Input Validation, Output Monitoring
Hardware Layer: TEEs, Secure Boot, PUFs
Plan for Runtime Adaptation and Recovery
Monitoring, Logging, Alerting
Rollback, Isolation, Incident Response
Design Feedback Loop

Figure 15.17: A design flow for building secure and trustworthy ML systems.

This design flow emphasizes the importance of a comprehensive approach to security, where each layer of the system is fortified against potential threats while remaining adaptable to evolving risks. By integrating these principles into the design and deployment of machine learning systems, organizations can build solutions that are not only effective but also resilient, trustworthy, and aligned with ethical standards.

15.8 Offensive Capabilities

While machine learning systems are often treated as assets to protect, they may also serve as tools for launching attacks. In adversarial settings, the same models used to enhance productivity, automate perception, or assist decision-making can be repurposed to execute or amplify offensive operations. This dual-use characteristic of machine learning, its capacity to secure systems as well as to subvert them, marks a fundamental shift in how ML must be considered within system-level threat models.

An offensive use of machine learning refers to any scenario in which a machine learning model is employed to facilitate the compromise of another system. In such cases, the model itself is not the object under attack, but the mechanism through which an adversary advances their objectives. These applications may involve reconnaissance, inference, subversion, impersonation, or the automation of exploit strategies that would otherwise require manual execution.

Importantly, such offensive applications are not speculative. Attackers are already integrating machine learning into their toolchains across a wide range of activities, from spam filtering evasion to model, driven malware generation. What distinguishes these scenarios is the deliberate use of learning-based systems to extract, manipulate, or generate information in ways that undermine the confidentiality, integrity, or availability of targeted components.

To clarify the diversity and structure of these applications, Table 15.8 summarizes several representative use cases. For each, the table identifies the type of machine learning model typically employed, the underlying system vulnerability it exploits, and the primary advantage conferred by the use of machine learning.

Table 15.8: Offensive machine learning use cases.

Offensive Use Case	ML Model Type	Targeted System Vulnerability	Advantage of ML
Phishing and Social Engineering	Large Language Models (LLMs)	Human perception and communication systems	Personalized, context-aware message crafting
Reconnaissance and Fingerprinting	Supervised classifiers, clustering models	System configuration, network behavior	Scalable, automated profiling of system behavior
Exploit Generation	Code generation models, fine-tuned transformers	Software bugs, insecure code patterns	Automated discovery of candidate exploits
Data Extraction (Inference Attacks)	Classification models, inversion models	Privacy leakage through model outputs	Inference with limited or black-box access
Evasion of Detection Systems	Adversarial input generators	Detection boundaries in deployed ML systems	Crafting minimally perturbed inputs to evade filters
Hardware-Level Attacks	CNNs and RNNs for time-series analysis	Physical side-channels (e.g., power, timing, EM)	Learning leakage patterns directly from raw signals

Each of these scenarios illustrates how machine learning models can serve as amplifiers of adversarial capability. For example, language models enable more convincing and adaptable phishing attacks, while clustering and classification algorithms facilitate reconnaissance by learning system-level behavioral patterns. Similarly, adversarial example generators and inference models systematically uncover weaknesses in decision boundaries or data privacy protections, often requiring only limited external access to deployed systems. In hardware contexts, as discussed in the next section, deep neural networks trained on side-channel data can automate the extraction of cryptographic secrets from physical measurements—transforming an expert-driven process into a learnable pattern recognition task.

Although these applications differ in technical implementation, they share a common foundation: the adversary replaces a static exploit with a learned model capable of approximating or adapting to the target's vulnerable behavior. This shift increases flexibility, reduces manual overhead, and improves robustness in the face of evolving or partially obscured defenses.

What makes this class of threats particularly significant is their favorable scaling behavior. Just as accuracy in computer vision or language modeling improves with additional data, larger architectures, and greater compute resources, so too does the performance of attack-oriented machine learning models. A model trained on larger corpora of phishing attempts or power traces, for instance, may generalize more effectively, evade more detectors, or require fewer inputs to succeed. The same ecosystem that drives innovation in beneficial AI, including public datasets, open-source tooling, and scalable infrastructure, also lowers the barrier to developing effective offensive models.

This dynamic creates an asymmetry between attacker and defender. While defensive measures are bounded by deployment constraints, latency budgets, and regulatory requirements, attackers can scale training pipelines with minimal marginal cost. The widespread availability of pretrained models and public ML platforms further reduces the expertise required to develop high-impact attacks.

As a result, any comprehensive treatment of machine learning system security must consider not only the vulnerabilities of ML systems themselves but also the ways in which machine learning can be harnessed to compromise other components—whether software, data, or hardware. Understanding the offensive potential of machine-learned systems is essential for designing resilient, trustworthy, and forward-looking defenses.

15.8.1 Case Study: Deep Learning for SCA

One of the most well-known and reproducible demonstrations of deep-learning-assisted SCA is the SCAAML framework (Side-Channel Attacks Assisted with Machine Learning) (Bursztein et al. 2019). Developed by researchers at Google, SCAAML provides a practical implementation of the attack pipeline described above.

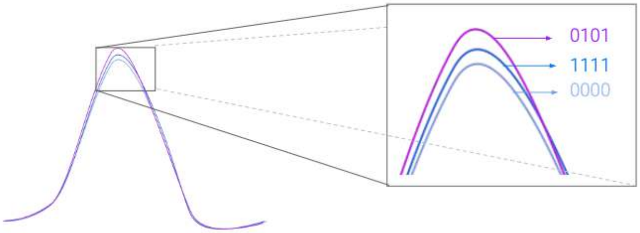


Figure 15.18: Power traces from an AES S-box operation. Source: Bursztein et al. (2019).

As shown in Figure 15.18, cryptographic computations exhibit data-dependent variations in their power consumption. These variations, while subtle, are measurable and reflect the internal state of the algorithm at specific points in time.

In traditional side-channel attacks, experts rely on statistical techniques to extract these differences. However, a neural network can learn to associate the shape of these signals with the specific data values being processed, effectively learning to decode the signal in a manner that mimics expert-crafted models, yet with enhanced flexibility and generalization. The model is trained on labeled examples of power traces and their corresponding intermediate values (e.g., output of an S-box operation[8]). Over time, it learns to associate patterns in the trace, similar to those depicted in Figure 15.18, with secret-dependent computational behavior. This transforms the key recovery task into a classification problem, where the goal is to infer the correct key byte based on trace shape alone.

In their study, Bursztein et al. (2019) trained a convolutional neural network to extract AES keys from power traces collected on an STM32F415 microcontroller

[8] The S-box (Substitution box) is a core component of the AES encryption algorithm that performs a non-linear substitution on each byte of data. During encryption, each input byte is replaced with a corresponding output value from a fixed lookup table. This operation is designed to provide confusion by creating a complex relationship between the key and ciphertext. The S-box operation's power consumption varies depending on the input value, making it a common target for side-channel analysis.

running the open-source TinyAES implementation. The model was trained to predict intermediate values of the AES algorithm, such as the output of the S-box in the first round, directly from raw power traces. Remarkably, the trained model was able to recover the full 128-bit key using only a small number of traces per byte.

The traces were collected using a ChipWhisperer setup with a custom STM32F target board, shown in Figure 15.19. This board executes AES operations while allowing external equipment to monitor power consumption with high temporal precision. The experimental setup captures how even inexpensive, low-power embedded devices can leak information through side channels—information that modern machine learning models can learn to exploit.

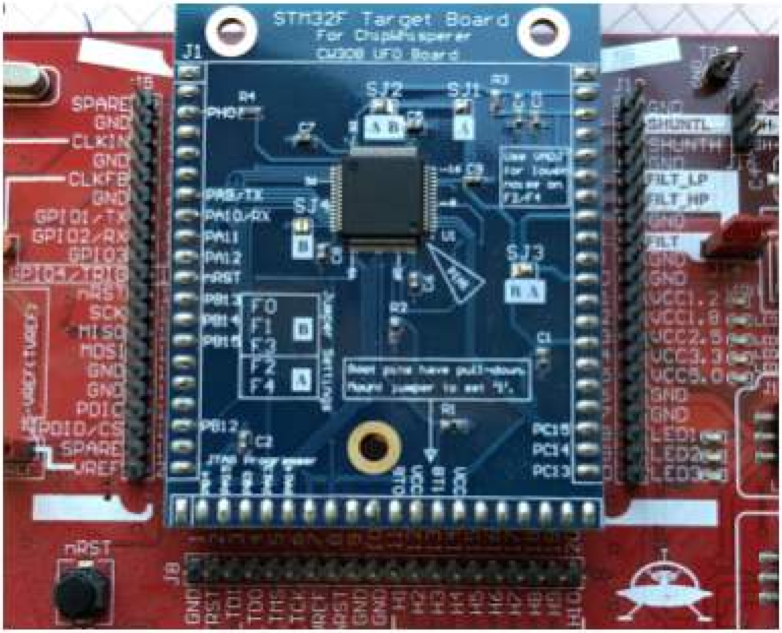


Figure 15.19: STM32F415 target board used in SCAAML experiments. Source: Bursztein et al. (2019).

Subsequent work expanded on this approach by introducing long-range models capable of leveraging broader temporal dependencies in the traces, improving performance even under noise and desynchronization (Bursztein et al. 2024). These developments highlight the potential for machine learning models to serve as offensive cryptanalysis tools—especially in the analysis of secure hardware.

The implications extend beyond academic interest. As deep learning models continue to scale, their application to side-channel contexts is likely to lower the cost, skill threshold, and trace requirements of hardware-level attacks—posing a growing challenge for the secure deployment of embedded machine learning systems, cryptographic modules, and trusted execution environments.

15.9 Conclusion

Security and privacy are foundational to the deployment of machine learning systems in real-world environments. As ML moves beyond the lab and into production, as it is deployed across cloud services, edge devices, mobile platforms, and critical infrastructure, the threats it faces become more complex and more consequential. From model theft and data leakage to adversarial manipulation and hardware compromise, securing ML systems requires a comprehensive understanding of the entire software and hardware stack.

This chapter explored these challenges from multiple angles. We began by examining real-world security incidents and threat models that impact ML systems, including attacks on training data, inference pipelines, and deployed models. We then discussed defense strategies that operate at different layers of the system: from data privacy techniques like differential privacy and federated learning, to robust model design, secure deployment practices, runtime monitoring, and hardware-enforced trust. Each of these layers addresses a distinct surface of vulnerability, and together they form the basis of a defense-in-depth approach.

Importantly, security is not a static checklist. It is an evolving process shaped by the deployment context, the capabilities of adversaries, and the risk tolerance of stakeholders. What protects a publicly exposed API may not suffice for an embedded medical device or a distributed fleet of autonomous systems. The effectiveness of any given defense depends on how well it fits into the larger system and how it interacts with other components, users, and constraints.

The goal of this chapter was not to catalog every threat or prescribe a fixed set of solutions. Rather, it was to help build the mindset needed to design secure, private, and trustworthy ML systems—systems that perform reliably under pressure, protect the data they rely on, and respond gracefully when things go wrong.

As we look ahead, security and privacy will remain intertwined with other system concerns: robustness, fairness, sustainability, and operational scale. In the chapters that follow, we will explore these additional dimensions and extend the foundation laid here toward the broader challenge of building ML systems that are not only performant, but responsible, reliable, and resilient by design.

15.10 Resources

ℹ Slides

- *Coming soon.*

! Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 16

Responsible AI

Figure 16.1: *DALL·E 3 Prompt: Illustration of responsible AI in a futuristic setting with the universe in the backdrop: A human hand or hands nurturing a seedling that grows into an AI tree, symbolizing a neural network. The tree has digital branches and leaves, resembling a neural network, to represent the interconnected nature of AI. The background depicts a future universe where humans and animals with general intelligence collaborate harmoniously. The scene captures the initial nurturing of the AI as a seedling, emphasizing the ethical development of AI technology in harmony with humanity and the universe.*

Purpose

How do human values translate into machine learning systems architecture, and what principles enable responsible system behavior at scale?

Machine learning systems do not exist in isolation—they operate within social, economic, and technical environments where their outputs affect people and institutions. As these systems grow in capability and reach, questions of responsibility become central to their design. The integration of fairness, transparency, and accountability is not an afterthought but a systems-level constraint that shapes data pipelines, model architectures, and deployment strategies. Recognizing the moral dimension of engineering choices is essential for building machine learning systems that serve human needs, avoid harm, and support long-term trust in automation.

Learning Objectives

- Grasp the foundational principles of responsible AI.
- Understand how responsible AI principles shape the design and operation of machine learning systems.
- Recognize the societal, organizational, and deployment contexts that influence responsible AI implementation.
- Identify tradeoffs and system-level challenges that arise when integrating ethical considerations into ML system design.
- Appreciate the role of governance, human oversight, and value alignment in sustaining trustworthy AI over time.

16.1 Overview

Machine learning systems are increasingly deployed in high-stakes domains such as healthcare, criminal justice, and employment. As their influence expands, so do the risks of embedding bias, compromising privacy, and enabling unintended harms. For example, a loan approval model trained exclusively on data from high-income neighborhoods may unfairly penalize applicants from underrepresented communities, reinforcing structural inequities.

Definition of Responsible AI

Responsible AI is the development and deployment of machine learning systems that explicitly uphold *ethical principles*, minimize *harm*, and promote *socially beneficial outcomes*. These systems treat *fairness*, *transparency*, *accountability*, *privacy*, and *safety* as *design constraints*, rather than afterthoughts, integrating them across the *machine learning lifecycle*.

Responsible machine learning aims to mitigate such outcomes by integrating ethical principles, such as fairness, transparency, accountability, and safety, into system design and operation. Fairness seeks to prevent discriminatory outcomes; explainability allows practitioners and users to interpret model behavior; robustness helps defend against adversarial manipulation and edge-case failures; and thorough validation supports trustworthy deployment.

Implementing these principles presents deep technical and organizational challenges. Engineers must grapple with mathematically defining fairness, reconciling competing objectives such as accuracy versus interpretability, and ensuring representative and reliable data pipelines. At the same time, institutions must align policies, incentives, and governance frameworks to uphold ethical development and deployment practices.

This chapter provides the foundations for understanding and implementing responsible machine learning. By examining the technical methods, design trade-offs, and broader system implications, it equips you to critically evaluate

AI systems and contribute to their development in a way that advances both capability and human values.

16.2 Core Principles

Responsible AI refers to the development and deployment of machine learning systems that intentionally uphold ethical principles and promote socially beneficial outcomes. These principles serve not only as policy ideals but as concrete constraints on system design, implementation, and governance.

Fairness refers to the expectation that machine learning systems do not discriminate against individuals or groups on the basis of protected attributes such as race, gender, or socioeconomic status. This principle encompasses both statistical metrics and broader normative concerns about equity, justice, and structural bias. The two key statistical measures of fairness are demographic parity and equalized odds. Demographic parity ensures equal outcomes across different demographic groups. For example, if a loan approval system maintains the same approval rate for all racial groups, it would satisfy demographic parity. The equalized odds criterion requires that equal outcomes be maintained for all groups at all decision thresholds. In practice, this means the true positive and false positive rates should be equal across protected groups. However, fairness extends beyond these statistical definitions to address deeper questions of equity, historical discrimination, and systemic bias in how machine learning systems impact different communities.

Explainability concerns the ability of stakeholders to interpret how a model produces its outputs. This involves understanding both how individual decisions are made and the model's overall behavior patterns. Explanations may be generated after a decision is made (called post-hoc explanations) to detail the reasoning process, or they may be built into the model's design for transparent operation. Explainability is essential for error analysis, regulatory compliance, and building user trust.

Transparency refers to openness about how AI systems are built, trained, validated, and deployed. It includes disclosure of data sources, design assumptions, system limitations, and performance characteristics. While explainability focuses on understanding outputs, transparency addresses the broader lifecycle of the system.

Accountability denotes the mechanisms by which individuals or organizations are held responsible for the outcomes of AI systems. It involves traceability, documentation, auditing, and the ability to remedy harms. Accountability ensures that AI failures are not treated as abstract malfunctions but as consequences with real-world impact.

Value Alignment is the principle that AI systems should pursue goals that are consistent with human intent and ethical norms. In practice, this involves both technical challenges, including reward design and constraint specification, and broader questions about whose values are represented and enforced.

Human Oversight emphasizes the role of human judgment in supervising, correcting, or halting automated decisions. This includes humans-in-the-loop during operation, as well as organizational structures that ensure AI use remains accountable to societal values and real-world complexity.

Other essential principles such as privacy and robustness are discussed in dedicated chapters on Security and Privacy and Robust AI, where their technical implementations and risks are explored in greater depth.

16.3 Princples in Practice

Responsible machine learning begins with a set of foundational principles, including fairness, transparency, accountability, privacy, and safety, that define what it means for an AI system to behave ethically and predictably. These principles are not abstract ideals or afterthoughts; they must be translated into concrete constraints that guide how models are trained, evaluated, deployed, and maintained.

Each principle sets expectations for system behavior. Fairness addresses how models treat different subgroups and respond to historical biases. Explainability ensures that model decisions can be understood by developers, auditors, and end users. Privacy governs what data is collected and how it is used. Accountability defines how responsibilities are assigned, tracked, and enforced throughout the system lifecycle. Safety requires that models behave reliably even in uncertain or shifting environments.

Taken together, these principles define what it means for a machine learning system to behave responsibly, not as isolated features but as system-level constraints that are embedded across the lifecycle. Table 16.1 provides a structured view of how key principles, including fairness, explainability, transparency, privacy, accountability, and robustness, map to the major phases of ML system development: data collection, model training, evaluation, deployment, and monitoring. While some principles (like fairness and privacy) begin with data, others (like robustness and accountability) become most critical during deployment and oversight. Explainability, though often emphasized during evaluation and user interaction, also supports model debugging and design-time validation. This table reinforces that responsible AI is not a post hoc consideration but a multi-phase architectural commitment.

Table 16.1: Responsible AI principles mapped to stages of the ML system lifecycle.

Principle	Data Collection	Model Training	Evaluation	Deployment	Monitoring
Fairness	Subgroup balance, representative data	Fairness-aware loss, reweighting	Group metrics, parity tests	Threshold tuning, access policy	Drift tracking by subgroup
Explainability	N/A	N/A	Global explanations, model inspection	Local explanations, user interfaces	Debug logs, recourse interfaces
Transparency	Data provenance, labeling standards	Training logs, documented assumptions	Report cards, test summaries	Model cards, known limitations	Version tracking, audit trails
Privacy	Consent protocols, data minimization	Differential privacy (e.g., DP-SGD)	N/A	Local inference, secure access	PII-free logging, auditability
Accountability	Data governance policies	Version control, loss attribution	Evaluation traceability	Decision override, usage documentation	Incident response, accountability triggers

Principle	Data Collection	Model Training	Evaluation	Deployment	Monitoring
Robustness	Diverse inputs, outlier flagging	Adversarial training, regularization	Stress tests, calibration	Fallback logic, abstention thresholds	Distribution shift alerts, failure modes

16.3.1 Transparency and Explainability

Machine learning systems are frequently criticized for their lack of interpretability. In many cases, models operate as opaque "black boxes,"[0] producing outputs that are difficult for users, developers, and regulators to understand or scrutinize. This opacity presents a significant barrier to trust, particularly in high-stakes domains such as criminal justice, healthcare, and finance, where accountability and the right to recourse are essential. For example, the COMPAS algorithm, used in the United States to assess recidivism risk, was found to exhibit racial bias. However, the proprietary nature of the system, combined with limited access to interpretability tools, hindered efforts to investigate or address the issue.

Explainability is the capacity to understand how a model produces its predictions. It includes both *local explanations*, which clarify individual predictions, and *global explanations*, which describe the models general behavior. Transparency, by contrast, encompasses openness about the broader system design and operation. This includes disclosure of data sources, feature engineering, model architectures, training procedures, evaluation protocols, and known limitations. Transparency also involves documentation of intended use cases, system boundaries, and governance structures.

These principles are not merely best practices; in many jurisdictions, they are legal obligations. For instance, the European Unions General Data Protection Regulation (GDPR)[1] requires that individuals receive meaningful information about the logic of automated decisions that significantly affect them. Similar regulatory pressures are emerging in other domains, reinforcing the need to treat explainability and transparency as core architectural requirements.

In practice, implementing these principles entails anticipating the needs of different stakeholders. Developers require diagnostic access to model internals; domain experts seek interpretable summaries of outputs; regulators and auditors demand clear documentation and traceability; and end users expect understandable justifications for system behavior. Designing for explainability and transparency therefore necessitates decisions about how and where to surface relevant information across the system lifecycle.

These principles also support system reliability over time. As models are retrained or updated, mechanisms for interpretability and traceability enable the detection of unexpected behavior, enable root cause analysis, and support governance. Transparency and explainability, when embedded into the structure and operation of a system, provide the foundation for trust, oversight, and alignment with institutional and societal expectations.

16.3.2 Fairness in Machine Learning

Fairness in machine learning refers to the principle that automated systems should not disproportionately disadvantage individuals or groups on the basis of protected attributes such as race, gender, age, or socioeconomic status.

[0] Black Box: Term in machine learning referring to systems where the decision process is not visible or well-understood by users.

[1] GDPR: European regulation mandating transparency about personal data use and the logic of automated decisions.

Because these systems are trained on historical data, they are susceptible to reproducing and amplifying patterns of systemic bias[2] embedded in that data. Without careful design, machine learning systems may unintentionally reinforce social inequities rather than mitigate them.

A widely studied example comes from the healthcare domain. An algorithm used to allocate care management resources in U.S. hospitals was found to systematically underestimate the health needs of Black patients (Obermeyer et al. 2019). The model used healthcare expenditures as a proxy for health status, but due to longstanding disparities in access and spending, Black patients were less likely to incur high costs. As a result, the model inferred that they were less sick, despite often having equal or greater medical need. This case illustrates how seemingly neutral design choices such as proxy variable[3] selection can yield discriminatory outcomes when historical inequities are not properly accounted for.

To evaluate fairness, a range of formal criteria have been developed that quantify how models perform across groups defined by sensitive attributes. Suppose a model $h(x)$ predicts a binary outcome, such as loan repayment, and let S represent a sensitive attribute with subgroups a and b. Several widely used fairness definitions are:

16.3.2.1 Demographic Parity

This criterion requires that the probability of receiving a positive prediction is independent of group membership. Formally, the model satisfies demographic parity if:

$$P(h(x) = 1 \mid S = a) = P(h(x) = 1 \mid S = b)$$

This means the model assigns favorable outcomes, such as loan approval or treatment referral, at equal rates across subgroups defined by a sensitive attribute S.

In the healthcare example, demographic parity would ask whether Black and white patients were referred for care at the same rate, regardless of their underlying health needs. While this might seem fair in terms of equal access, it ignores real differences in medical status and risk, potentially overcorrecting in situations where needs are not evenly distributed.

16.3.2.2 Equalized Odds

This definition requires that the model's predictions are conditionally independent of group membership given the true label. Specifically, the true positive and false positive rates must be equal across groups:

$$P(h(x) = 1 \mid S = a, Y = y) = P(h(x) = 1 \mid S = b, Y = y), \quad \text{for } y \in \{0, 1\}.$$

That is, for each true outcome $Y = y$, the model should produce the same prediction distribution across groups $S = a$ and $S = b$. This means the model should behave similarly across groups for individuals with the same true

[2] Systemic Bias: Deep-rooted bias in societal structures, often unconsciously perpetuated.

[3] Proxy Variable: A variable used to indirectly represent another where direct measures are unavailable.

outcome—whether they qualify for a positive result or not. It ensures that errors (both missed and incorrect positives) are distributed equally.

Applied to the medical case, equalized odds would ensure that patients with the same actual health needs (the true label Y) are equally likely to be correctly or incorrectly referred, regardless of race. The original algorithm violated this by under-referring Black patients who were equally or more sick than their white counterparts—highlighting unequal true positive rates.

16.3.2.3 Equality of Opportunity

A relaxation of equalized odds, this criterion focuses only on the true positive rate. It requires that, among individuals who should receive a positive outcome, the probability of receiving one is equal across groups:

$$P(h(x) = 1 \mid S = a, Y = 1) = P(h(x) = 1 \mid S = b, Y = 1).$$

This ensures that qualified individuals, who have $Y = 1$, are treated equally by the model regardless of group membership.

In our running example, this measure would ensure that among patients who do require care, both Black and white individuals have an equal chance of being identified by the model. In the case of the U.S. hospital system, the algorithm's use of healthcare expenditure as a proxy variable led to a failure in meeting this criterion—Black patients with significant health needs were less likely to receive care due to their lower historical spending.

These definitions capture different aspects of fairness and are generally incompatible. Satisfying one may preclude satisfying another, reflecting the reality that fairness involves tradeoffs between competing normative goals. Determining which metric to prioritize requires careful consideration of the application context, potential harms, and stakeholder values (Barocas, Hardt, and Narayanan 2023).

In operational systems, fairness must be treated as a constraint that informs decisions throughout the machine learning lifecycle. It is shaped by how data are collected and represented, how objectives and proxies are selected, how model predictions are thresholded, and how feedback mechanisms are structured. For example, a choice between ranking versus classification models can yield different patterns of access across groups, even when using the same underlying data.

While fairness metrics help formalize equity goals, they are often limited to predefined demographic categories. In practice, these categories may be too coarse to capture the full range of disparities present in real-world data. A principled approach to fairness must account for overlapping and intersectional identities, ensuring that model behavior remains consistent across subgroups that may not be explicitly labeled in advance. Recent work in this area emphasizes the need for predictive reliability across a wide range of population slices (Hébert-Johnson et al. 2018), reinforcing the idea that fairness must be considered a system-level requirement, not a localized adjustment. This expanded view of fairness highlights the importance of designing architectures, evaluation protocols, and monitoring strategies that support more nuanced, context-sensitive assessments of model behavior.

Ultimately, fairness is a system-wide property that arises from the interaction of data engineering practices, modeling choices, evaluation procedures, and decision policies. It cannot be isolated to a single model component or resolved through post hoc adjustments alone. Responsible machine learning design requires treating fairness as a foundational constraint—one that informs architectural choices, workflows, and governance mechanisms throughout the entire lifecycle of the system.

16.3.3 Privacy and Data Governance

Machine learning systems often rely on extensive collections of personal data to support model training and enable personalized functionality. This reliance introduces significant responsibilities related to user privacy, data protection, and ethical data stewardship. Responsible AI design treats privacy not as an ancillary feature, but as a fundamental constraint that must inform decisions across the entire system lifecycle.

One of the core challenges in supporting privacy is the inherent tension between data utility and individual protection. Rich, high-resolution datasets can enhance model accuracy and adaptability but also heighten the risk of exposing sensitive information—particularly when datasets are aggregated or linked with external sources. For example, models trained on conversational data or medical records have been shown to memorize specific details that can later be retrieved through model queries or adversarial interaction (Ippolito et al. 2023).

Even seemingly innocuous data can produce privacy risks when combined. Wearable devices that track physiological and behavioral signals, including heart rate, movement, or location, may individually seem benign but can jointly reveal detailed user profiles. These risks are further exacerbated when users have limited visibility or control over how their data is processed, retained, or transmitted.

Privacy as a system principle also entails robust data governance. This includes defining what data is collected, under what conditions, and with what degree of consent and transparency. Responsible governance requires attention to labeling practices, access controls, logging infrastructure, and compliance with jurisdictional requirements. These mechanisms serve to constrain how data flows through a system and to document accountability for its use.

To support structured decision-making in this space, Figure 16.2 shows a simplified flowchart outlining key privacy checkpoints in the early stages of a data pipeline. It highlights where core safeguards, such as consent acquisition, encryption, and differential privacy, should be applied. While actual implementations often involve more nuanced tradeoffs and context-sensitive decisions, this diagram provides a scaffold for identifying where privacy risks arise and how they can be mitigated through responsible design choices.

The consequences of weak data governance are well documented. Systems trained on poorly understood or biased datasets may perpetuate structural inequities or expose sensitive attributes unintentionally. In the COMPAS example introduced earlier, the lack of transparency surrounding data provenance and usage precluded effective evaluation or redress. In clinical applications,

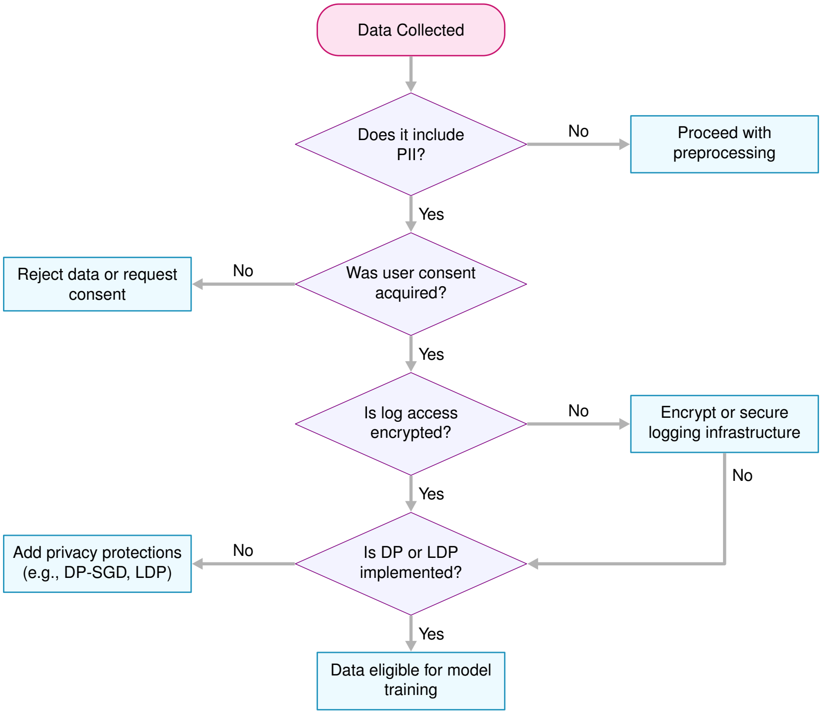

Data Collected
Does it include PII?
No
Proceed with preprocessing
Yes
Was user consent acquired?
No
Reject data or request consent
Yes
Is log access encrypted?
No
Encrypt or secure logging infrastructure
No
Yes
Is DP or LDP implemented?
No
Add privacy protections (e.g., DP-SGD, LDP)
Yes
Data eligible for model training

Figure 16.2: A simplified flowchart illustrating decision points in a privacy-aware data pipeline.

datasets frequently reflect artifacts such as missing values or demographic skew that compromise both performance and privacy. Without clear standards for data quality and documentation, such vulnerabilities become systemic.

Privacy is not solely the concern of isolated algorithms or data processors—it must be addressed as a structural property of the system. Decisions about consent collection, data retention, model design, and auditability all contribute to the privacy posture of a machine learning pipeline. This includes the need to anticipate risks not only during training, but also during inference and ongoing operation. Threats such as membership inference attacks underscore the importance of embedding privacy safeguards into both model architecture and interface behavior.

Legal frameworks increasingly reflect this understanding. Regulations such as the GDPR, CCPA, and APPI impose specific obligations regarding data minimization, purpose limitation, user consent, and the right to deletion. These requirements translate ethical expectations into enforceable design constraints, reinforcing the need to treat privacy as a core principle in system development.

Ultimately, privacy in machine learning is a system-wide commitment. It requires coordination across technical and organizational domains to ensure that data usage aligns with user expectations, legal mandates, and societal norms. Rather than viewing privacy as a constraint to be balanced against functionality, responsible system design integrates privacy from the outset—

informing architecture, shaping interfaces, and constraining how models are built, updated, and deployed.

16.3.4 Designing for Safety and Robustness

Safety in machine learning refers to the assurance that models behave predictably under normal conditions and fail in controlled, non-catastrophic ways under stress or uncertainty. Closely related, robustness concerns a models ability to maintain stable and consistent performance in the presence of variation—whether in inputs, environments, or system configurations. Together, these properties are foundational for responsible deployment in safety-critical domains, where machine learning outputs directly affect physical or high-stakes decisions.

Ensuring safety and robustness in practice requires anticipating the full range of conditions a system may encounter and designing for behavior that remains reliable beyond the training distribution. This includes not only managing the variability of inputs but also addressing how models respond to unexpected correlations, rare events, and deliberate attempts to induce failure. For example, widely publicized failures in autonomous vehicle systems have revealed how limitations in object detection or overreliance on automation can result in harmful outcomes—even when models perform well under nominal test conditions.

One illustrative failure mode arises from adversarial inputs: carefully constructed perturbations that appear benign to humans but cause a model to output incorrect or harmful predictions (Szegedy et al. 2013b). Such vulnerabilities are not limited to image classification—they have been observed across modalities including audio, text, and structured data, and they reveal the brittleness of learned representations in high-dimensional spaces. These behaviors highlight that robustness must be considered not only during training but as a global property of how systems interact with real-world complexity.

A related challenge is distribution shift: the inevitable mismatch between training data and conditions encountered in deployment. Whether due to seasonality, demographic changes, sensor degradation, or environmental variability, such shifts can degrade model reliability even in the absence of adversarial manipulation. Failures under distribution shift may propagate through downstream decisions, introducing safety risks that extend beyond model accuracy alone. In domains such as healthcare, finance, or transportation, these risks are not hypothetical—they carry real consequences for individuals and institutions.

Responsible machine learning design treats robustness as a systemic requirement. Addressing it requires more than improving individual model performance. It involves designing systems that anticipate uncertainty, surface their limitations, and support fallback behavior when predictive confidence is low. This includes practices such as setting confidence thresholds, supporting abstention from decision-making, and integrating human oversight into operational workflows. These mechanisms are critical for building systems that degrade gracefully rather than failing silently or unpredictably.

Safety and robustness also impose requirements at the architectural and organizational level. Decisions about how models are monitored, how failures are detected, and how updates are governed all influence whether a system can respond effectively to changing conditions. Responsible design demands that robustness be treated not as a property of isolated models but as a constraint that shapes the overall behavior of machine learning systems.

16.3.5 Accountability and Governance

Accountability in machine learning refers to the capacity to identify, attribute, and address the consequences of automated decisions. It extends beyond diagnosing failures to ensuring that responsibility for system behavior is clearly assigned, that harms can be remedied, and that ethical standards are maintained through oversight and institutional processes. Without such mechanisms, even well-intentioned systems can generate significant harm without recourse, undermining public trust and eroding legitimacy.

Unlike traditional software systems, where responsibility often lies with a clearly defined developer or operator, accountability in machine learning is distributed. Model outputs are shaped by upstream data collection, training objectives, pipeline design, interface behavior, and post-deployment feedback. These interconnected components often involve multiple actors across technical, legal, and organizational domains. For example, if a hiring platform produces biased outcomes, accountability may rest not only with the model developer but also with data providers, interface designers, and deploying institutions. Responsible system design requires that these relationships be explicitly mapped and governed.

Inadequate governance can prevent institutions from recognizing or correcting harmful model behavior. The failure of Google Flu Trends[4] to anticipate distribution shift and feedback loops illustrates how opacity in model assumptions and update policies can inhibit corrective action. Without visibility into the system's design and data curation, external stakeholders lacked the means to evaluate its validity, contributing to the model's eventual discontinuation.

Legal frameworks increasingly reflect the necessity of accountable design. Regulations such as the Illinois Artificial Intelligence Video Interview Act and the EU AI Act impose requirements for transparency, consent, documentation, and oversight in high-risk applications. These policies embed accountability not only in the outcomes a system produces, but in the operational procedures and documentation that support its use. Internal organizational changes, including the introduction of fairness audits and the imposition of usage restrictions in targeted advertising systems, demonstrate how regulatory pressure can catalyze structural reforms in governance.

Designing for accountability entails supporting traceability at every stage of the system lifecycle. This includes documenting data provenance, recording model versioning, enabling human overrides, and retaining sufficient logs for retrospective analysis. Tools such as model cards and datasheets for datasets exemplify practices that make system behavior interpretable and reviewable. However, accountability is not reducible to documentation alone—it also requires mechanisms for feedback, contestation, and redress.

Within organizations, governance structures help formalize this responsibility. Ethics review processes, cross-functional audits, and model risk committees

[4] Google Flu Trends was a service launched in 2008 that attempted to predict flu outbreaks by analyzing Google search queries. It was discontinued in 2015 after consistently overestimating flu prevalence, which highlights the risks of relying on indirect behavioral data without robust validation.

provide forums for anticipating downstream impact and responding to emerging concerns. These structures must be supported by infrastructure that allows users to contest decisions and developers to respond with corrections. For instance, systems that enable explanations or user-initiated reviews help bridge the gap between model logic and user experience, especially in domains where the impact of error is significant.

Architectural decisions also play a role. Interfaces can be designed to surface uncertainty, enable escalation, or suspend automated actions when appropriate. Logging and monitoring pipelines must be configured to detect signs of ethical drift[5], such as performance degradation across subpopulations or unanticipated feedback loops. In distributed systems, where uniform observability is difficult to maintain, accountability must be embedded through architectural safeguards—such as secure protocols, update constraints, or trusted components.

Governance does not imply centralized control. Instead, it involves distributing responsibility in ways that are transparent, actionable, and sustainable. Technical teams, legal experts, end users, and institutional leaders must all have access to the tools and information necessary to evaluate system behavior and intervene when necessary. As machine learning systems become more complex and embedded in critical infrastructure, accountability must scale accordingly—becoming a foundational consideration in both architecture and process, not a reactive layer added after deployment.

16.4 Deployment Contexts

Responsible AI principles, such as fairness, privacy, transparency, and robustness, cannot be implemented uniformly across all system architectures. Their realization is shaped by the constraints and affordances of the deployment environment. A model operating in a centralized cloud setting benefits from high computational capacity, centralized monitoring, and scalable retraining pipelines, but may introduce substantial privacy and governance risks. In contrast, systems deployed on mobile devices, edge platforms, or embedded microcontrollers face stringent constraints on latency, memory, energy, and connectivity—factors that directly affect how responsible AI can be supported in practice.

These architectural differences introduce tradeoffs that affect not only what is technically feasible, but also how responsibilities are distributed across system components. Resource availability, latency constraints, user interface design, and the presence or absence of connectivity all play a role in determining whether responsible AI principles can be enforced consistently across deployment contexts.

Understanding how deployment shapes the operational landscape for fairness, explainability, safety, privacy, and accountability is essential for designing machine learning systems that are robust, aligned, and sustainable across real-world settings.

[5] Ethical Drift: The gradual deterioration of ethical standards in automated systems, often due to new or shifting contexts and overlooked biases.

16.4.1 System Explainability

The feasibility of explainability in machine learning systems is deeply shaped by deployment context. While model architecture and explanation technique are important factors, system-level constraints, including computational capacity, latency requirements, interface design, and data accessibility, determine whether interpretability can be supported in a given environment. These constraints vary significantly across cloud platforms, mobile devices, edge systems, and deeply embedded deployments, affecting both the form and timing of explanations.

In high-resource environments, such as centralized cloud systems, techniques like SHAP[6] and LIME[7] can be used to generate detailed post hoc explanations, even if they require multiple forward passes or sampling procedures. These methods are often impractical in latency-sensitive or resource-constrained settings, where explanation must be lightweight and fast. On mobile devices or embedded systems, methods based on saliency maps or input gradients are more feasible, as they typically involve a single backward pass. In TinyML deployments, runtime explanation may be infeasible altogether, making development-time inspection the primary opportunity for ensuring interpretability.

Latency and interactivity also influence the delivery of explanations. In real-time systems, such as drones or automated industrial control loops, there may be no opportunity to present or compute explanations during operation. Logging internal signals or confidence scores for later analysis becomes the primary strategy. In contrast, systems with asynchronous interactions, such as financial risk scoring or medical diagnosis, allow for deeper and delayed explanations to be rendered after the decision has been made.

Audience requirements further shape design choices. End users typically require explanations that are concise, intuitive, and contextually meaningful. For instance, a mobile health app might summarize a prediction as “elevated heart rate during sleep,” rather than referencing abstract model internals. By contrast, developers, auditors, and regulators often need access to attribution maps[8], concept activations, or decision traces to perform debugging, validation, or compliance review. These internal explanations must be exposed through developer-facing interfaces or embedded within the model development workflow.

Explainability also varies across the system lifecycle. During model development, interpretability supports diagnostics, feature auditing, and concept verification. After deployment, explainability shifts toward runtime behavior monitoring, user communication, and post hoc analysis of failure cases. In systems where runtime explanation is infeasible, such as in TinyML—design-time validation becomes especially critical, requiring models to be constructed in a way that anticipates and mitigates downstream interpretability failures.

Treating explainability as a system design constraint means planning for interpretability from the outset. It must be balanced alongside other deployment requirements, including latency budgets, energy constraints, and interface limitations. Responsible system design allocates sufficient resources—not only for predictive performance, but for ensuring that stakeholders can meaningfully

[6] SHAP (SHapley Additive exPlanations): A game theoretic approach to explain the output of any machine learning model by calculating the contribution of each feature to the prediction.

[7] LIME (Local Interpretable Model-agnostic Explanations): A method to explain predictions by learning an interpretable model locally around the prediction by perturbing the input and observing changes in model output.

[8] Attribution Maps: Visuals or data that map out the contribution of each part of the input data to the final decision of a model.

understand and evaluate model behavior within the operational limits of the deployment environment.

16.4.2 Fairness Constraints

While fairness can be formally defined, its operationalization is shaped by deployment-specific constraints. Differences in data access, model personalization, computational capacity, and infrastructure for monitoring or retraining affect how fairness can be evaluated, enforced, and sustained across diverse system architectures.

A key determinant is data visibility. In centralized environments, such as cloud-hosted platforms, developers often have access to large datasets with demographic annotations. This enables the use of group-level fairness metrics, fairness-aware training procedures, and post hoc auditing. In contrast, decentralized deployments, such as federated learning clients or mobile applications, typically lack access to global statistics due to privacy constraints or fragmented data. In such settings, fairness interventions must often be embedded during training or dataset curation, as post-deployment evaluation may be infeasible.

Personalization and adaptation mechanisms also influence fairness tradeoffs. Systems that deliver a global model to all users may target parity across demographic groups. In contrast, locally adapted models such as those embedded in health monitoring apps or on-device recommendation engines may aim for individual fairness, ensuring consistent treatment of similar users. However, enforcing this is challenging in the absence of clear similarity metrics or representative user data. Furthermore, personalized systems that retrain based on local behavior may drift toward reinforcing existing disparities, particularly when data from marginalized users is sparse or noisy.

Real-time and resource-constrained environments impose additional limitations. Embedded systems, wearables, or real-time control platforms often cannot support runtime fairness monitoring or dynamic threshold adjustment. In these scenarios, fairness must be addressed proactively through conservative design choices, including balanced training objectives and static evaluation of subgroup performance prior to deployment. For example, a speech recognition system deployed on a low-power wearable may need to ensure robust performance across different accents at design time, since post-deployment recalibration is not possible.

Decision thresholds and system policies also affect realized fairness. Even when a model performs similarly across groups, applying a uniform threshold across all users may lead to disparate impacts if score distributions differ. A mobile loan approval system, for instance, may systematically under-approve one group unless group-specific thresholds are considered. Such decisions must be explicitly reasoned about, justified, and embedded into the systems policy logic in advance of deployment.

Long-term fairness is further shaped by feedback dynamics. Systems that retrain on user behavior, including ranking models, recommender systems, and automated decision pipelines, may reinforce historical biases unless feedback loops are carefully managed. For example, a hiring platform that disproportionately favors candidates from specific institutions may amplify existing inequalities when retrained on biased historical outcomes. Mitigating such effects requires governance mechanisms that span not only training but also deployment monitoring, data logging, and impact evaluation.

Fairness, like other responsible AI principles, is not confined to model parameters or training scripts. It emerges from a series of decisions across the full system lifecycle: data acquisition, model design, policy thresholds, retraining infrastructure, and user feedback handling. Treating fairness as a system-level constraint, particularly in constrained or decentralized deployments, requires anticipating where tradeoffs may arise and ensuring that fairness objectives are embedded into architecture, decision rules, and lifecycle management from the outset.

16.4.3 Privacy Architectures

Privacy in machine learning systems is not confined to protecting individual records; it is shaped by how data is collected, stored, transmitted, and integrated into system behavior. These decisions are tightly coupled to deployment architecture. System-level privacy constraints vary widely depending on whether a model is hosted in the cloud, embedded on-device, or distributed across user-controlled environments—each presenting different challenges for minimizing risk while maintaining functionality.

A key architectural distinction is between centralized and decentralized data handling. Centralized cloud systems typically aggregate data at scale, enabling high-capacity modeling and monitoring. However, this aggregation increases exposure to breaches and surveillance, making strong encryption, access control, and auditability essential. In decentralized deployments, including mobile applications, federated learning clients, and TinyML systems, data remains local, reducing central risk but limiting global observability. These environments often prevent developers from accessing the demographic or behavioral statistics needed to monitor system performance or enforce compliance, requiring privacy safeguards to be embedded during development.

Privacy challenges are especially pronounced in systems that personalize behavior over time. Applications such as smart keyboards, fitness trackers, or voice assistants continuously adapt to users by processing sensitive signals like location, typing patterns, or health metrics. Even when raw data is discarded, trained models may retain user-specific patterns that can be recovered via inference-time queries. In architectures where memory is persistent and interaction is frequent, managing long-term privacy requires tight integration of protective mechanisms into the model lifecycle.

Connectivity assumptions further shape privacy design. Cloud-connected systems enable centralized enforcement of encryption protocols and remote deletion policies, but may introduce latency, energy overhead, or increased exposure during data transmission. In contrast, edge systems typically operate offline or intermittently, making privacy enforcement dependent on architectural constraints—such as feature minimization, local data retention, and compile-time obfuscation. On TinyML devices, which often lack persistent storage or update channels, privacy must be engineered into the static firmware and model binaries, leaving no opportunity for post-deployment adjustment.

Privacy risks also extend to the serving and monitoring layers. A model with logging enabled, or one that updates through active learning, may inadvertently expose sensitive information if logging infrastructure is not privacy-aware. For example, membership inference attacks can reveal whether a users data was included in training by analyzing model outputs. Defending against such attacks requires that privacy-preserving measures extend beyond training and into interface design, rate limiting, and access control.

Crucially, privacy is not determined solely by technical mechanisms but by how users experience the system. A model may meet formal privacy definitions and still violate user expectations if data collection is opaque or explanations are lacking. Interface design plays a central role: systems must clearly communicate what data is collected, how it is used, and how users can opt out or revoke consent. In privacy-sensitive applications, failure to align with user norms can erode trust even in technically compliant systems.

Architectural decisions thus influence privacy at every stage of the data lifecycle—from acquisition and preprocessing to inference and monitoring. Designing for privacy involves not only choosing secure algorithms, but also making principled tradeoffs based on deployment constraints, user needs, and legal obligations. In high-resource settings, this may involve centralized enforcement and policy tooling. In constrained environments, privacy must be embedded statically in model design and system behavior, often without the possibility of dynamic oversight.

Privacy is not a feature to be appended after deployment. It is a system-level property that must be planned, implemented, and validated in concert with the architectural realities of the deployment environment.

16.4.4 Safety and Robustness

The implementation of safety and robustness in machine learning systems is fundamentally shaped by deployment architecture. Systems deployed in dynamic, unpredictable environments, including autonomous vehicles, healthcare robotics, and smart infrastructure, must manage real-time uncertainty and mitigate the risk of high-impact failures. Others, such as embedded controllers or on-device ML systems, require stable and predictable operation under resource constraints, limited observability, and restricted opportunities for recovery. In all cases, safety and robustness are system-level properties that depend not only on model quality, but on how failures are detected, contained, and managed in deployment.

One recurring challenge is distribution shift: when conditions at deployment diverge from those encountered during training. Even modest shifts in input characteristics, including lighting, sensor noise, or environmental variability, can significantly degrade performance if uncertainty is not modeled or monitored. In architectures lacking runtime monitoring or fallback mechanisms, such degradation may go undetected until failure occurs. Systems intended for real-world variability must be architected to recognize when inputs fall outside expected distributions and to either recalibrate or defer decisions accordingly.

Adversarial robustness introduces an additional set of architectural considerations. In systems that make security-sensitive decisions, including fraud detection, content moderation, and biometric verification, adversarial inputs can compromise reliability. Mitigating these threats may involve both model-level defenses (e.g., adversarial training, input filtering) and deployment-level strategies, such as API access control, rate limiting, or redundancy in input validation. These protections often impose latency and complexity tradeoffs that must be carefully balanced against real-time performance requirements.

Latency-sensitive deployments further constrain robustness strategies. In autonomous navigation, real-time monitoring, or control systems, decisions must be made within strict temporal budgets. Heavyweight robustness mechanisms may be infeasible, and fallback actions must be defined in advance. Many such systems rely on confidence thresholds, abstention logic, or rule-based overrides to reduce risk. For example, a delivery robot may proceed only when pedestrian detection confidence is high enough; otherwise, it pauses or defers to human oversight. These control strategies often reside outside the learned model, but must be tightly integrated into the systems safety logic.

TinyML deployments introduce additional constraints. Deployed on microcontrollers with minimal memory, no operating system, and no connectivity, these systems cannot rely on runtime monitoring or remote updates. Safety and robustness must be engineered statically through conservative design, extensive pre-deployment testing, and the use of models that are inherently simple and predictable. Once deployed, the system must operate reliably under conditions such as sensor degradation, power fluctuations, or environmental variation—without external intervention or dynamic correction.

Across all deployment contexts, monitoring and escalation mechanisms are essential for sustaining robust behavior over time. In cloud or high-resource settings, systems may include uncertainty estimators, distributional change detectors, or human-in-the-loop feedback loops to detect failure conditions and trigger recovery. In more constrained settings, these mechanisms must be simplified or precomputed, but the principle remains: robustness is not achieved once, but maintained through the ongoing ability to recognize and respond to emerging risks.

Safety and robustness must be treated as emergent system properties. They depend on how inputs are sensed and verified, how outputs are acted upon, how failure conditions are recognized, and how corrective measures are initiated. A robust system is not one that avoids all errors, but one that fails visibly, controllably, and safely. In safety-critical applications, designing for this behavior is not optional—it is a foundational requirement.

16.4.5 Governance Structures

Accountability in machine learning systems must be realized through concrete architectural choices, interface designs, and operational procedures. Governance structures make responsibility actionable by defining who is accountable for system outcomes, under what conditions, and through what mechanisms. These structures are deeply influenced by deployment architecture. The degree to which accountability can be traced, audited, and enforced varies across centralized, mobile, edge, and embedded environments—each posing distinct challenges for maintaining system oversight and integrity.

In centralized systems, such as cloud-hosted platforms, governance is typically supported by robust infrastructure for logging, version control, and real-time monitoring. Model registries, telemetry dashboards, and structured event pipelines allow teams to trace predictions to specific models, data inputs, or configuration states. This visibility enables accountability to be distributed across development and operations teams, and to be institutionalized through impact assessments, fairness audits, or regulatory compliance workflows. However, the scale and complexity of such systems, which often comprise hundreds of models that serve diverse users, can obscure failure pathways and complicate attribution.

In contrast, edge deployments distribute intelligence to devices that may operate independently from centralized infrastructure. Embedded models in vehicles, factories, or homes must support localized mechanisms for detecting abnormal behavior, triggering alerts, and escalating issues. For example, an industrial sensor might flag anomalies when its prediction confidence drops, initiating a predefined escalation process. Designing for such autonomy requires forethought: engineers must determine what signals to capture, how to store them locally, and how to reassign responsibility when connectivity is intermittent or delayed.

Mobile deployments, such as personal finance apps or digital health tools, exist at the intersection of user interfaces and backend systems. When something goes wrong, it is often unclear whether the issue lies with a local model, a remote service, or the broader design of the user interaction. Governance in these settings must account for this ambiguity. Effective accountability requires clear documentation, accessible recourse pathways, and mechanisms for surfacing, explaining, and contesting automated decisions at the user level. The ability to understand and appeal outcomes must be embedded into both the interface and the surrounding service architecture.

In TinyML deployments, governance is especially constrained. Devices may lack connectivity, persistent storage, or runtime configurability, limiting opportunities for dynamic oversight or intervention. Here, accountability must be embedded statically—through mechanisms such as cryptographic firmware signatures, fixed audit trails, and pre-deployment documentation of training data and model parameters. In some cases, governance must be enforced during manufacturing or provisioning, since no post-deployment correction is possible. These constraints make the design of governance structures inseparable from early-stage architectural decisions.

Interfaces also play a critical role in enabling accountability. Systems that surface explanations, expose uncertainty estimates, or allow users to query decision histories make it possible for developers, auditors, or users to understand both what occurred and why. By contrast, opaque APIs, undocumented thresholds, or closed-loop decision systems inhibit oversight. Effective governance requires that information flows be aligned with stakeholder needs, including technical, regulatory, and user-facing aspects, so that failure modes are observable and remediable.

Governance approaches must also adapt to domain-specific risks and institutional norms. High-stakes applications, such as healthcare or criminal justice, often involve legally mandated impact assessments and audit trails. Lower-risk domains may rely more heavily on internal practices, shaped by customer expectations, reputational concerns, or technical conventions. Regardless of the setting, governance must be treated as a system-level design property—not an external policy overlay. It is implemented through the structure of codebases, deployment pipelines, data flows, and decision interfaces.

Sustaining accountability across diverse deployment environments requires planning not only for success, but for failure. This includes defining how anomalies are detected, how roles are assigned, how records are maintained, and how remediation occurs. These processes must be embedded in infrastructure—traceable in logs, enforceable through interfaces, and resilient to the architectural constraints of the systems deployment context.

16.4.6 Design Tradeoffs

Machine learning systems do not operate in idealized silos. Their deployment contexts, whether cloud based, mobile, edge deployed, or deeply embedded, impose competing constraints that shape how responsible AI principles can be realized. Tradeoffs arise not because ethical values are ignored, but because no deployment environment can simultaneously optimize for all objectives under finite resources, strict latency requirements, evolving user behavior, and regulatory complexity.

Cloud based systems often support extensive monitoring, fairness audits, interpretability services, and privacy preserving tools due to ample computational and storage resources. However, these benefits typically come with centralized data handling, which introduces risks related to surveillance, data breaches, and complex governance. In contrast, on device systems such as mobile applications, edge platforms, or TinyML deployments provide stronger data locality and user control, but limit post deployment visibility, fairness instrumentation, and model adaptation.

Tensions between goals often become apparent at the architectural level. For example, systems with real time response requirements, such as wearable gesture recognition or autonomous braking, cannot afford to compute detailed interpretability explanations during inference. Designers must choose whether to precompute simplified outputs, defer explanation to asynchronous analysis, or omit interpretability altogether in runtime settings.

Conflicts also emerge between personalization and fairness. Systems that adapt to individuals based on local usage data often lack the global context necessary to assess disparities across population subgroups. Ensuring that personalized predictions do not result in systematic exclusion requires careful architectural design, balancing user level adaptation with mechanisms for group level equity and auditability.

Privacy and robustness objectives can also conflict. Robust systems often benefit from logging rare events or user outliers to improve reliability. However, recording such data may conflict with privacy goals or violate legal constraints on data minimization. In settings where sensitive behavior must remain local or encrypted, robustness must be designed into the model architecture and training procedure in advance, since post hoc refinement may not be feasible.

These examples illustrate a broader systems level challenge. Responsible AI principles cannot be considered in isolation. They interact, and optimizing for one may constrain another. The appropriate balance depends on deployment architecture, stakeholder priorities, domain specific risks, and the consequences of error.

What distinguishes responsible machine learning design is not the elimination of tradeoffs, but the clarity and deliberateness with which they are navigated. Design decisions must be made transparently, with a full understanding of the limitations imposed by the deployment environment and the impacts of those decisions on system behavior.

These architectural tensions are summarized in Table 16.2, which compares how responsible AI principles manifest across cloud, mobile, edge, and TinyML systems. Each setting imposes different constraints on explainability, fairness, privacy, safety, and accountability, based on factors such as compute capacity, connectivity, data access, and governance feasibility.

Table 16.2: Comparison of key principles across Cloud, Edge, Mobile, and TinyML deployments.

Principle	Cloud ML	Edge ML	Mobile ML	TinyML
Explainability	Supports complex models and methods like SHAP and sampling approaches	Needs lightweight, low-latency methods like saliency maps	Requires interpretable outputs for users, often defers deeper analysis to the cloud	Severely limited due to constrained hardware; mostly static or compile-time only
Fairness	Large datasets enable bias detection and mitigation	Localized biases harder to detect but allows on-device adjustments	High personalization complicates group-level fairness tracking	Minimal data limits bias analysis and mitigation
Privacy	Centralized data at risk of breaches but can utilize strong encryption and differential privacy methods	Sensitive personal data on-device requires on-device protections	Tight coupling to user identity requires consent-aware design and local processing	Distributed data reduces centralized risks but poses challenges for anonymization
Safety	Vulnerable to hacking and large-scale attacks	Real-world interactions make reliability critical	Operates under user supervision, but still requires graceful failure	Needs distributed safety mechanisms due to autonomy
Accountability	Corporate policies and audits enable traceability and oversight	Fragmented supply chains complicate accountability	Requires clear user-facing disclosures and feedback paths	Traceability required across long, complex hardware chains
Governance	External oversight and regulations like GDPR or CCPA are feasible	Requires self-governance by developers and integrators	Balances platform policy with app developer choices	Relies on built-in protocols and cryptographic assurances

The table highlights the importance of tailoring responsible AI strategies to the characteristics of the deployment environment. Across system types, core values must be implemented in ways that align with operational realities, regulatory obligations, and user expectations.

16.5 Technical Foundations

Responsible machine learning depends on technical methods that translate ethical principles into actionable mechanisms within system design. These methods show *how* to detect and mitigate bias, preserve user privacy, improve robustness, and support interpretability—not as abstract ideals, but as system behaviors that can be engineered, tested, and maintained. Their effectiveness depends not only on their theoretical properties, but on how well they align with practical constraints such as data quality, resource availability, user interaction models, and deployment architecture.

These methods are not interchangeable or universally applicable. Each introduces tradeoffs involving accuracy, latency, scalability, and implementation complexity. Choosing the right approach requires understanding the methods purpose, its assumptions, and the demands it places on the surrounding system. Moreover, technical interventions must be evaluated not just at the model level, but across the machine learning lifecycle, including data acquisition, training, deployment, monitoring, and updating.

This section presents representative techniques for operationalizing responsible AI principles in practice. Each method is introduced with attention to its role within the system, its typical use cases, and the architectural requirements it imposes. While no single method ensures responsible behavior in isolation, together these tools form the foundation for building machine learning systems that perform reliably and uphold societal and ethical expectations.

16.5.1 Bias Detection and Mitigation

Operationalizing fairness in deployed systems requires more than principled objectives or theoretical metrics—it demands system-aware methods that detect, measure, and mitigate bias across the machine learning lifecycle. Building on the system-level constraints discussed earlier, fairness must be treated as an architectural consideration that intersects with data engineering, model training, inference design, monitoring infrastructure, and policy governance. While fairness metrics such as demographic parity, equalized odds, and equality of opportunity formalize different normative goals, their realization depends on the architecture's ability to measure subgroup performance, support adaptive decision boundaries, and store or surface group-specific metadata during runtime.

Practical implementation is often shaped by limitations in data access and system instrumentation. In many real-world environments, especially in mobile, federated, or embedded systems, sensitive attributes such as gender, age, or race may not be available at inference time, making it difficult to track or audit model performance across demographic groups. In such contexts, fairness interventions must occur upstream during data curation or training, as post-deployment recalibration may not be feasible. Even when data is available, continuous retraining pipelines that incorporate user feedback can reinforce existing disparities unless explicitly monitored for fairness degradation. For example, an on-device recommendation model that adapts to user behavior may amplify prior biases if it lacks the infrastructure to detect demographic imbalances in user interactions or outputs.

Figure 16.3 illustrates how fairness constraints can introduce tension with deployment choices. In a binary loan approval system, two subgroups, Subgroup A, represented in blue, and Subgroup B, represented in red, require different decision thresholds to achieve equal true positive rates. Using a single threshold across groups leads to disparate outcomes, potentially disadvantaging Subgroup B. Addressing this imbalance by adjusting thresholds per group may improve fairness, but doing so requires support for conditional logic in the model serving stack, access to sensitive attributes at inference time, and a governance framework for explaining and justifying differential treatment across groups.

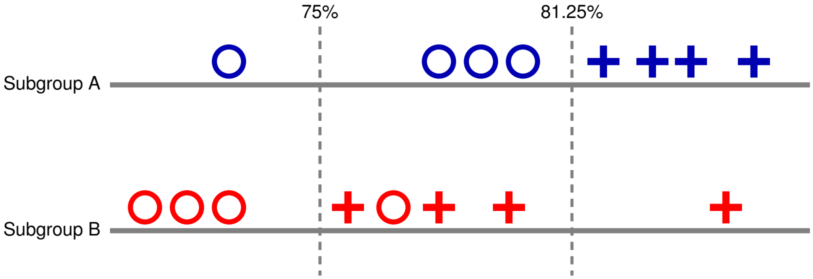


Figure 16.3: Illustrates the trade-off in setting classification thresholds for two subgroups (A and B) in a loan repayment model. Plusses (+) represent true positives (repayers), and circles (O) represent true negatives (defaulters). Different thresholds (75% for B and 81.25% for A) maximize subgroup accuracy but reveal fairness challenges.

Fairness interventions may be applied at different points in the pipeline, but each comes with system-level implications. Preprocessing methods, which rebalance training data through sampling, reweighting, or augmentation, require access to raw features and group labels, often through a feature store or data lake that preserves lineage. These methods are well-suited to systems with centralized training pipelines and high-quality labeled data. In contrast, in-processing approaches embed fairness constraints directly into the optimization objective. These require training infrastructure that can support custom loss functions or constrained solvers and may demand longer training cycles or additional regularization validation.

Post-processing methods, including the application of group-specific thresholds or the adjustment of scores to equalize outcomes, require inference systems that can condition on sensitive attributes or reference external policy rules. This demands coordination between model serving infrastructure, access control policies, and logging pipelines to ensure that differential treatment is both auditable and legally defensible. Moreover, any post-processing strategy must be carefully validated to ensure that it does not compromise user experience, model stability, or compliance with jurisdictional regulations on attribute use.

Scalable fairness enforcement often requires more advanced strategies, such as multicalibration, which ensures that model predictions remain calibrated across a wide range of intersecting subgroups (Hébert-Johnson et al. 2018). Implementing multicalibration at scale requires infrastructure for dynamically generating subgroup partitions, computing per-group calibration error, and integrating fairness audits into automated monitoring systems. These capabilities are typically only available in large-scale, cloud-based deployments with mature observability and metrics pipelines. In constrained environments such as embedded or TinyML systems, where telemetry is limited and model logic is fixed, such techniques are not feasible and fairness must be validated entirely at design time.

Across deployment environments, maintaining fairness requires lifecycle-aware mechanisms. Model updates, feedback loops, and interface designs all affect how fairness evolves over time. A fairness-aware model may degrade if retraining pipelines do not include fairness checks, if logging systems cannot track subgroup outcomes, or if user feedback introduces subtle biases not captured by training distributions. Monitoring systems must be equipped to surface fairness regressions, and retraining protocols must have access to subgroup-labeled validation data, which may require data governance policies and ethical review.

Fairness is not a one-time optimization, nor is it a property of the model in isolation. It emerges from coordinated decisions across data acquisition, feature engineering, model design, thresholding, feedback handling, and system monitoring. Embedding fairness into machine learning systems requires architectural foresight, operational discipline, and tooling that spans the full deployment stack—from training workflows to serving infrastructure to user-facing interfaces.

16.5.2 Privacy Preservation

Recall that privacy is a foundational principle of responsible machine learning, with implications that extend across data collection, model behavior, and user interaction. Privacy constraints are shaped not only by ethical and legal obligations, but also by the architectural properties of the system and the context in which it is deployed. Technical methods for privacy preservation aim to prevent data leakage, limit memorization, and uphold user rights such as consent, opt-out, and data deletion—particularly in systems that learn from personalized or sensitive information.

Modern machine learning models, especially large-scale neural networks, are known to memorize individual training examples, including names, locations, or excerpts of private communication (Ippolito et al. 2023). This memorization presents significant risks in privacy-sensitive applications such as smart assistants, wearables, or healthcare platforms, where training data may encode protected or regulated content. For example, a voice assistant that adapts to user speech may inadvertently retain specific phrases, which could later be extracted through carefully designed prompts or queries.

This risk is not limited to language models. As shown in Figure 16.4, diffusion models trained on image datasets have been observed to regenerate visual instances from the training set. Such behavior highlights a more general vulnerability: many contemporary model architectures can internalize and reproduce training data, often without explicit signals or intent, and without easy detection or control.

Beyond memorization, models are susceptible to membership inference attacks, in which adversaries attempt to determine whether a specific datapoint was part of the training set (Shokri et al. 2017). These attacks exploit subtle differences in model behavior between seen and unseen inputs. In high-stakes applications such as healthcare or legal prediction, the mere knowledge that an individuals record was used in training may violate privacy expectations or regulatory requirements.

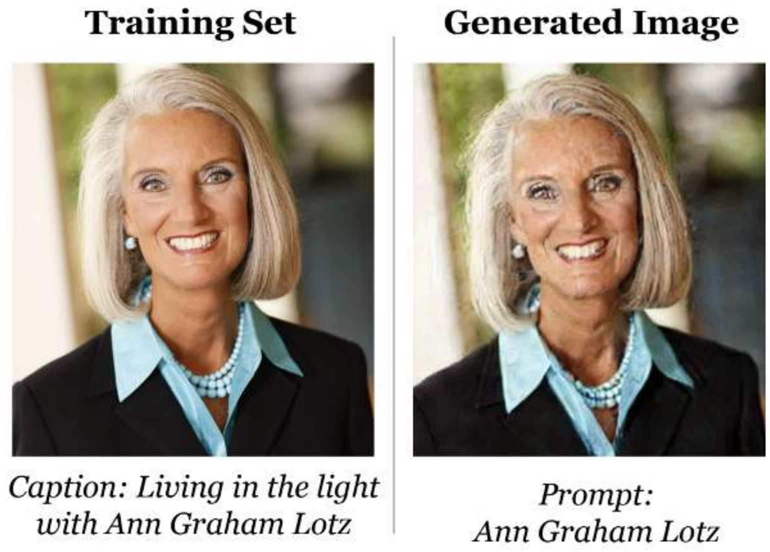


Figure 16.4: Diffusion models memorizing samples from training data. Source: Ippolito et al. (2023).

To mitigate such vulnerabilities, a range of privacy-preserving techniques have been developed. Among the most widely adopted is differential privacy, which provides formal guarantees that the inclusion or exclusion of a single datapoint has a statistically bounded effect on the models output. Algorithms such as differentially private stochastic gradient descent (DP-SGD)[9] enforce these guarantees by clipping gradients and injecting noise during training (Martin Abadi et al. 2016). When implemented correctly, these methods prevent the model from memorizing individual datapoints and reduce the risk of inference attacks.

However, differential privacy introduces significant system-level tradeoffs. The noise added during training can degrade model accuracy, increase the number of training iterations, and require access to larger datasets to maintain performance. These constraints are especially pronounced in resource-limited deployments such as mobile, edge, or embedded systems, where memory, compute, and power budgets are tightly constrained. In such settings, it may be necessary to combine lightweight privacy techniques (e.g., feature obfuscation, local differential privacy) with architectural strategies that limit data collection, shorten retention, or enforce strict access control at the edge.

Privacy enforcement also depends on infrastructure beyond the model itself. Data collection interfaces must support informed consent and transparency. Logging systems must avoid retaining sensitive inputs unless strictly necessary, and must support access controls, expiration policies, and auditability. Model serving infrastructure must be designed to prevent overexposure of outputs that could leak internal model behavior or allow reconstruction of private data. These system-level mechanisms require close coordination between ML engineering, platform security, and organizational governance.

[9] Differentially Private Stochastic Gradient Descent (DP-SGD): An algorithm that uses gradient clipping and noise injection to preserve privacy during training.

Moreover, privacy must be enforced not only during training but throughout the machine learning lifecycle. Retraining pipelines must account for deleted or revoked data, especially in jurisdictions with data deletion mandates. Monitoring infrastructure must avoid recording personally identifiable information in logs or dashboards. Privacy-aware telemetry collection, secure enclave deployment, and per-user audit trails are increasingly used to support these goals, particularly in applications with strict legal oversight.

Architectural decisions also vary by deployment context. Cloud-based systems may rely on centralized enforcement of differential privacy, encryption, and access control, supported by telemetry and retraining infrastructure. In contrast, edge and TinyML systems must build privacy constraints into the deployed model itself, often with no runtime configurability or feedback channel. In such cases, static analysis, conservative design, and embedded privacy guarantees must be implemented at compile time, with validation performed prior to deployment.

Ultimately, privacy is not an attribute of a model in isolation but a system-level property that emerges from design decisions across the pipeline. Responsible privacy preservation requires that technical safeguards, interface controls, infrastructure policies, and regulatory compliance mechanisms work together to minimize risk throughout the lifecycle of a deployed machine learning system.

16.5.3 Machine Unlearning

Privacy preservation does not end at training time. In many real-world systems, users must retain the right to revoke consent or request the deletion of their data, even after a model has been trained and deployed. Supporting this requirement introduces a core technical challenge: how can a model "forget" the influence of specific datapoints without requiring full retraining—a task that is often infeasible in edge, mobile, or embedded deployments with constrained compute, storage, and connectivity?

Traditional approaches to data deletion assume that the full training dataset remains accessible and that models can be retrained from scratch after removing the targeted records. Figure 16.5 contrasts traditional model retraining with emerging machine unlearning approaches. While retraining involves reconstructing the model from scratch using a modified dataset, unlearning aims to remove a specific datapoint's influence without repeating the entire learning process.

This distinction becomes critical in systems with tight latency, compute, or privacy constraints. These assumptions rarely hold in practice. Many deployed machine learning systems do not retain raw training data due to security, compliance, or cost constraints. In such environments, full retraining is often impractical and operationally disruptive, especially when data deletion must be verifiable, repeatable, and audit-ready.

Machine unlearning aims to address this limitation by removing the influence of individual datapoints from an already trained model without retraining it entirely. Current approaches approximate this behavior by adjusting internal parameters, modifying gradient paths, or isolating and pruning components of the model so that the resulting predictions reflect what would have been

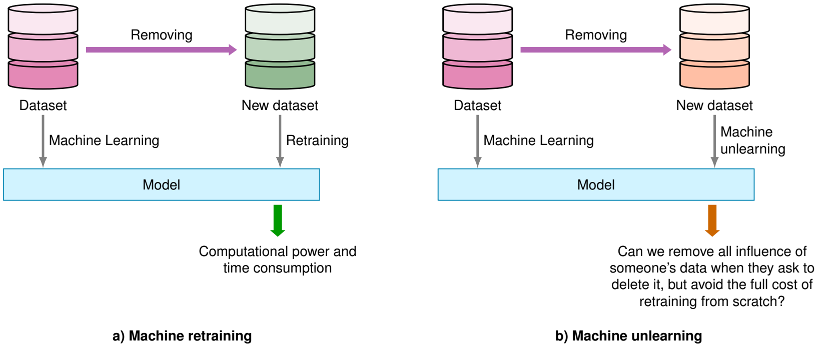

Removing
Dataset
New dataset
Machine Learning
Retraining
Model
Computational power and time consumption
a) Machine retraining
Removing
Dataset
New dataset
Machine Learning
Machine unlearning
Model
Can we remove all influence of someone's data when they ask to delete it, but avoid the full cost of retraining from scratch?
b) Machine unlearning

Figure 16.5: Machine retraining versus unlearning.

learned without the deleted data (Bourtoule et al. 2021). These techniques are still maturing and may require simplified model architectures, additional tracking metadata, or compromise on model accuracy and stability. They also introduce new burdens around verification: how to prove that deletion has occurred in a meaningful way, especially when internal model state is not fully interpretable.

The motivation for machine unlearning is reinforced by regulatory frameworks. Laws such as the General Data Protection Regulation (GDPR), the California Consumer Privacy Act (CCPA), and similar statutes in Canada and Japan codify the right to be forgotten, including for data used in model training. These laws increasingly require not just prevention of unauthorized data access, but proactive revocation—empowering users to request that their information cease to influence downstream system behavior. High-profile incidents in which generative models have reproduced personal content or copyrighted data highlight the practical urgency of integrating unlearning mechanisms into responsible system design.

From a systems perspective, machine unlearning introduces nontrivial architectural and operational requirements. Systems must be able to track data lineage, including which datapoints contributed to a given model version. This often requires structured metadata capture and training pipeline instrumentation. Additionally, systems must support user-facing deletion workflows, including authentication, submission, and feedback on deletion status. Verification may require maintaining versioned model registries, along with mechanisms for confirming that the updated model exhibits no residual influence from the deleted data. These operations must span data storage, training orchestration, model deployment, and auditing infrastructure, and they must be robust to failure or rollback.

These challenges are amplified in resource-constrained deployments. TinyML systems typically run on devices with no persistent storage, no connectivity, and highly compressed models. Once deployed, they cannot be updated or retrained in response to deletion requests. In such settings, machine unlearning is effectively infeasible post-deployment and must be enforced during initial model development through static data minimization and conservative generalization strategies. Even in cloud-based systems, where retraining is more tractable, unlearning must contend with distributed training pipelines, replication across services, and the difficulty of synchronizing deletion across model snapshots and logs.

Despite these challenges, machine unlearning is becoming essential for responsible system design. As machine learning systems become more embedded, personalized, and adaptive, the ability to revoke training influence becomes central to maintaining user trust and meeting legal requirements. Critically, unlearning cannot be retrofitted after deployment. It must be considered during the architecture and policy design phases, with support for lineage tracking, re-training orchestration, and deployment roll-forward built into the system from the beginning.

Machine unlearning represents a shift in privacy thinking—from protecting what data is collected, to controlling how long that data continues to affect system behavior. This lifecycle-oriented perspective introduces new challenges for model design, infrastructure planning, and regulatory compliance, while also providing a foundation for more user-controllable, transparent, and adaptable machine learning systems.

16.5.4 Adversarial Robustness

Machine learning models, particularly deep neural networks, are known to be vulnerable to small, carefully crafted perturbations that significantly alter their predictions. These vulnerabilities, first formalized through the concept of adversarial examples (Szegedy et al. 2013b), highlight a gap between model performance on curated training data and behavior under real-world variability. A model that performs reliably on clean inputs may fail when exposed to inputs that differ only slightly from its training distribution—differences imperceptible to humans, but sufficient to change the model's output.

This phenomenon is not limited to theory. Adversarial examples have been used to manipulate real systems, including content moderation pipelines (Bhagoji et al. 2018), ad-blocking detection (Tramèr et al. 2019), and voice recognition models (Carlini et al. 2016). In safety-critical domains such as autonomous driving or medical diagnostics, even rare failures can have high-consequence outcomes, compromising user trust or opening attack surfaces for malicious exploitation.

Figure 16.6 illustrates a visually negligible perturbation that causes a confident misclassification—underscoring how subtle changes can produce disproportionately harmful effects.

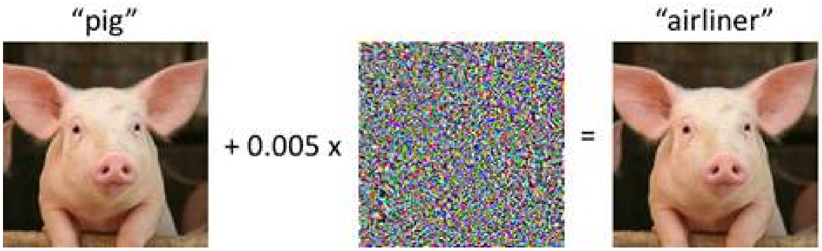


Figure 16.6: Perturbation effect on prediction. Source: Microsoft.

At its core, adversarial vulnerability stems from an architectural mismatch between model assumptions and deployment conditions. Many training pipelines assume data is clean, independent, and identically distributed. In contrast, deployed systems must operate under uncertainty, noise, domain shift, and possible adversarial tampering. Robustness, in this context, encompasses not only the ability to resist attack but also the ability to maintain consistent behavior under degraded or unpredictable conditions.

Improving robustness begins at training. Adversarial training, one of the most widely used techniques, augments training data with perturbed examples. This helps the model learn more stable decision boundaries but typically increases training time and reduces clean-data accuracy. Implementing adversarial training at scale also places demands on data preprocessing pipelines, model checkpointing infrastructure, and validation protocols that can accommodate perturbed inputs.

Architectural modifications can also promote robustness. Techniques that constrain a models Lipschitz constant[10], regularize gradient sensitivity, or enforce representation smoothness can make predictions more stable. These design changes must be compatible with the models expressive needs and the underlying training framework. For example, smooth models may be preferred for embedded systems with limited input precision or where safety-critical thresholds must be respected.

At inference time, systems may implement uncertainty-aware decision-making. Models can abstain from making predictions when confidence is low, or route uncertain inputs to fallback mechanisms—such as rule-based components or human-in-the-loop systems. These strategies require deployment infrastructure that supports fallback logic, user escalation workflows, or configurable abstention policies. For instance, a mobile diagnostic app might return "inconclusive" if model confidence falls below a specified threshold, rather than issuing a potentially harmful prediction.

Monitoring infrastructure plays a critical role in maintaining robustness post-deployment. Distribution shift detection, anomaly tracking, and behavior drift analytics allow systems to identify when robustness is degrading over time. Implementing these capabilities requires persistent logging of model inputs, predictions, and contextual metadata, as well as secure channels for triggering retraining or escalation. These tools introduce their own systems overhead and must be integrated with telemetry services, alerting frameworks, and model versioning workflows.

Beyond empirical defenses, formal approaches offer stronger guarantees. Certified defenses, such as randomized smoothing[11], provide probabilistic assurances that a models output will remain stable within a bounded input region. These methods require multiple forward passes per inference and are computationally intensive, making them suitable primarily for high-assurance, resource-rich environments. Their integration into production workflows also demands compatibility with model serving infrastructure and probabilistic verification tooling.

Simpler defenses, such as input preprocessing, filter inputs through denoising, compression, or normalization steps to remove adversarial noise. These transformations must be lightweight enough for real-time execution, especially

[10] Lipschitz Constant: A measure that quantifies the sensitivity of the output of a function to changes in its input.

[11] Randomized Smoothing: A technique used to create a probabilistically robust version of a model against input perturbations.

in edge deployments, and robust enough to preserve task-relevant features. Another approach is ensemble modeling, in which predictions are aggregated across multiple diverse models. This increases robustness but adds complexity to inference pipelines, increases memory footprint, and complicates deployment and maintenance workflows.

System constraints such as latency, memory, power budget, and model update cadence strongly shape which robustness strategies are feasible. Adversarial training increases model size and training duration, which may challenge CI/CD pipelines and increase retraining costs. Certified defenses demand computational headroom and inference time tolerance. Monitoring requires logging infrastructure, data retention policies, and access control. On-device and TinyML deployments, in particular, often cannot accommodate runtime checks or dynamic updates. In such cases, robustness must be validated statically and embedded at compile time.

Ultimately, adversarial robustness is not a standalone model attribute. It is a system-level property that emerges from coordination across training, model architecture, inference logic, logging, and fallback pathways. A model that appears robust in isolation may still fail if deployed in a system that lacks monitoring or interface safeguards. Conversely, even a partially robust model can contribute to overall system reliability if embedded within an architecture that detects uncertainty, limits exposure to untrusted inputs, and supports recovery when things go wrong.

Robustness, like privacy and fairness, must be engineered not just into the model, but into the system surrounding it. Responsible ML system design requires anticipating the ways in which models might fail under real-world stress—and building infrastructure that makes those failures detectable, recoverable, and safe.

16.5.5 Explainability and Interpretability

As machine learning systems are deployed in increasingly consequential domains, the ability to understand and interpret model predictions becomes essential. Explainability and interpretability refer to the technical and design mechanisms that make a models behavior intelligible to human stakeholders—whether developers, domain experts, auditors, regulators, or end users. While the terms are often used interchangeably, interpretability typically refers to the inherent transparency of a model, such as a decision tree or linear classifier. Explainability, in contrast, encompasses techniques for generating post hoc justifications for predictions made by complex or opaque models.

Explainability plays a central role in system validation, error analysis, user trust, regulatory compliance, and incident investigation. In high-stakes domains such as healthcare, financial services, and autonomous decision systems, explanations help determine whether a model is making decisions for legitimate reasons or relying on spurious correlations. For instance, an explainability tool might reveal that a diagnostic model is overly sensitive to image artifacts rather than medical features, which is a failure mode that could otherwise go undetected. Regulatory frameworks in many sectors now mandate that

AI systems provide “meaningful information” about how decisions are made, reinforcing the need for systematic support for explanation.

Explainability methods can be broadly categorized based on when they operate and how they relate to model structure. Post hoc methods are applied after training and treat the model as a black box. These methods do not require access to internal model weights and instead infer influence patterns or feature contributions from model behavior. Common post hoc techniques include feature attribution methods such as input gradients, Integrated Gradients, GradCAM (Selvaraju et al. 2017), LIME (Ribeiro, Singh, and Guestrin 2016), and SHAP (Lundberg and Lee 2017). These approaches are widely used in image and tabular domains, where explanations can be rendered as saliency maps or feature rankings.

Another post hoc approach involves counterfactual explanations, which describe how a models output would change if the input were modified in specific ways. These are especially relevant for decision-facing applications such as credit or hiring systems. For example, a counterfactual explanation might state that an applicant would have received a loan approval if their reported income were higher or their debt lower (Wachter, Mittelstadt, and Russell 2017). Counterfactual generation requires access to domain-specific constraints and realistic data manifolds, making integration into real-time systems challenging.

A third class of techniques relies on concept-based explanations, which attempt to align learned model features with human-interpretable concepts. For example, a convolutional network trained to classify indoor scenes might activate filters associated with “lamp,” “bed,” or “bookshelf” (C. J. Cai et al. 2019). These methods are especially useful in domains where subject matter experts expect explanations in familiar semantic terms. However, they require training data with concept annotations or auxiliary models for concept detection, which introduces additional infrastructure dependencies.

While post hoc methods are flexible and broadly applicable, they come with limitations. Because they approximate reasoning after the fact, they may produce plausible but misleading rationales. Their effectiveness depends on model smoothness, input structure, and the fidelity of the explanation technique. These methods are often most useful for exploratory analysis, debugging, or user-facing summaries—not as definitive accounts of internal logic.

In contrast, inherently interpretable models are transparent by design. Examples include decision trees, rule lists, linear models with monotonicity constraints, and k-nearest neighbor classifiers. These models expose their reasoning structure directly, enabling stakeholders to trace predictions through a set of interpretable rules or comparisons. In regulated or safety-critical domains such as recidivism prediction or medical triage, inherently interpretable models may be preferred, even at the cost of some accuracy (Rudin 2019). However, these models generally do not scale well to high-dimensional or unstructured data, and their simplicity can limit performance in complex tasks.

The relative interpretability of different model types can be visualized along a spectrum. As shown in Figure 16.7, models such as decision trees and linear regression offer transparency by design, whereas more complex architectures like neural networks and convolutional models require external techniques to explain their behavior. This distinction is central to choosing an appropriate model for a given application—particularly in settings where regulatory scrutiny or stakeholder trust is paramount.

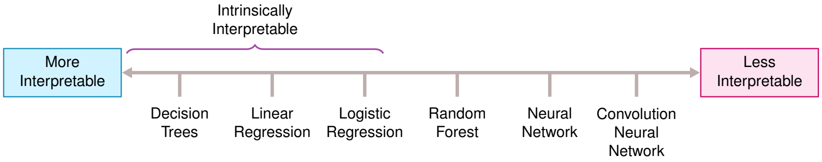


Figure 16.7: Spectrum of model interpretability. Inherently interpretable models (e.g., decision trees, linear regression) are transparent by design, while complex models (e.g., neural networks, convolutional models) require post hoc explanation techniques.

Hybrid approaches aim to combine the representational capacity of deep models with the transparency of interpretable components. Concept bottleneck models (Koh et al. 2020), for example, first predict intermediate, interpretable variables and then use a simple classifier to produce the final prediction. ProtoPNet models (C. Chen et al. 2019) classify examples by comparing them to learned prototypes, offering visual analogies for users to understand predictions. These hybrid methods are attractive in domains that demand partial transparency, but they introduce new system design considerations, such as the need to store and index learned prototypes and surface them at inference time.

A more recent research direction is mechanistic interpretability, which seeks to reverse-engineer the internal operations of neural networks. This line of work, inspired by program analysis and neuroscience, attempts to map neurons, layers, or activation patterns to specific computational functions (Olah et al. 2020; Geiger et al. 2021). Although promising, this field remains exploratory and is currently most relevant to the analysis of large foundation models where traditional interpretability tools are insufficient.

From a systems perspective, explainability introduces a number of architectural dependencies. Explanations must be generated, stored, surfaced, and evaluated within system constraints. The required infrastructure may include explanation APIs, memory for storing attribution maps, visualization libraries, and logging mechanisms that capture intermediate model behavior. Models must often be instrumented with hooks or configured to support repeated evaluations—particularly for explanation methods that require sampling, perturbation, or backpropagation.

These requirements interact directly with deployment constraints. For instance, methods like SHAP and LIME involve multiple forward passes or surrogate model fitting and may be impractical in latency-sensitive or resource-constrained environments such as edge devices or real-time decision systems. In such settings, systems may rely on approximations such as precomputed explanations, simplified attribution methods, or fallback rule-based logic. Explainability must also align with interface capabilities: wearable devices, for example, may support only brief textual or audio explanations, requiring designers to prioritize clarity and brevity.

Explainability spans the full machine learning lifecycle. During development, interpretability tools are used for dataset auditing, concept validation, and early debugging. At inference time, they support accountability, decision verification, and user communication. Post-deployment, explanations may be logged, surfaced in audits, or queried during error investigations. System design must support each of these phases—ensuring that explanation tools are integrated into training frameworks, model serving infrastructure, and user-facing applications.

Compression and optimization techniques also affect explainability. Pruning, quantization, and architectural simplifications often used in TinyML or mobile settings can distort internal representations or disable gradient flow, degrading the reliability of attribution-based explanations. In such cases, interpretability must be validated post-optimization to ensure that it remains meaningful and trustworthy. If explanation quality is critical, these transformations must be treated as part of the design constraint space.

Ultimately, explainability is not an add-on feature—it is a system-wide concern. Designing for interpretability requires careful decisions about who needs explanations, what kind of explanations are meaningful, and how those explanations can be delivered given the systems latency, compute, and interface budget. As machine learning becomes embedded in critical workflows, the ability to explain becomes a core requirement for safe, trustworthy, and accountable systems.

16.5.6 Model Performance Monitoring

Training-time evaluations, no matter how rigorous, do not guarantee reliable model performance once a system is deployed. Real-world environments are dynamic: input distributions shift due to seasonality, user behavior evolves in response to system outputs, and contextual expectations change with policy or regulation. These factors can cause predictive performance, and even more critically, system trustworthiness, to degrade over time. A model that performs well under training or validation conditions may still make unreliable or harmful decisions in production.

The implications of such drift extend beyond raw accuracy. Fairness guarantees may break down if subgroup distributions shift relative to the training set, or if features that previously correlated with outcomes become unreliable in new contexts. Interpretability demands may also evolve—for instance, as new stakeholder groups seek explanations, or as regulators introduce new transparency requirements. Trustworthiness, therefore, is not a static property conferred at training time, but a dynamic system attribute shaped by deployment context and operational feedback.

To ensure responsible behavior over time, machine learning systems must incorporate mechanisms for continual monitoring, evaluation, and corrective action. Monitoring involves more than tracking aggregate accuracy—it requires surfacing performance metrics across relevant subgroups, detecting shifts in input distributions, identifying anomalous outputs, and capturing meaningful user feedback. These signals must then be compared to predefined expectations around fairness, robustness, and transparency, and linked to actionable system responses such as model retraining, recalibration, or rollback.

Implementing effective monitoring depends on robust infrastructure. Systems must log inputs, outputs, and contextual metadata in a structured and secure manner. This requires telemetry pipelines that capture model versioning, input characteristics, prediction confidence, and post-inference feedback. These logs support drift detection and provide evidence for retrospective audits of fairness and robustness. Monitoring systems must also be integrated with alerting, update scheduling, and policy review processes to support timely and traceable intervention.

Monitoring also supports feedback-driven improvement. For example, repeated user disagreement, correction requests, or operator overrides can signal problematic behavior. This feedback must be aggregated, validated, and translated into updates to training datasets, data labeling processes, or model architecture. However, such feedback loops carry risks: biased user responses can introduce new inequities, and excessive logging can compromise privacy. Designing these loops requires careful coordination between user experience design, system security, and ethical governance.

Monitoring mechanisms vary by deployment architecture. In cloud-based systems, rich logging and compute capacity allow for real-time telemetry, scheduled fairness audits, and continuous integration of new data into retraining pipelines. These environments support dynamic reconfiguration and centralized policy enforcement. However, the volume of telemetry may introduce its own challenges in terms of cost, privacy risk, and regulatory compliance.

In mobile systems, connectivity is intermittent and data storage is limited. Monitoring must be lightweight and resilient to synchronization delays. Local inference systems may collect performance data asynchronously and transmit it in aggregate to backend systems. Privacy constraints are often stricter, particularly when personal data must remain on-device. These systems require careful data minimization and local aggregation techniques to preserve privacy while maintaining observability.

Edge deployments, such as those in autonomous vehicles, smart factories, or real-time control systems, demand low-latency responses and operate with minimal external supervision. Monitoring in these systems must be embedded within the runtime, with internal checks on sensor integrity, prediction confidence, and behavior deviation. These checks often require low-overhead implementations of uncertainty estimation, anomaly detection, or consistency validation. System designers must anticipate failure conditions and ensure that anomalous behavior triggers safe fallback procedures or human intervention.

TinyML systems, which operate on deeply embedded hardware with no connectivity, persistent storage, or dynamic update path, present the most constrained monitoring scenario. In these environments, monitoring must be designed and compiled into the system prior to deployment. Common strategies include input range checking, built-in redundancy, static failover logic, or conservative validation thresholds. Once deployed, these models operate independently, and any post-deployment failure may require physical device replacement or firmware-level reset.

Despite these differences, the core challenge is universal: deployed ML systems must not only perform well initially, but continue to behave responsibly as the environment changes. Monitoring provides the observability layer that links system performance to ethical goals and accountability structures. Without monitoring, fairness and robustness become invisible. Without feedback, misalignment cannot be corrected. Monitoring, therefore, is the operational foundation that enables machine learning systems to remain adaptive, auditable, and aligned with their intended purpose over time.

16.6 Sociotechnical and Ethical Systems Considerations

Responsible machine learning system design extends beyond technical correctness and algorithmic safeguards. Once deployed, these systems operate within complex sociotechnical environments where their outputs influence, and are influenced by, human behavior, institutional practices, and evolving societal norms. Over time, machine learning systems become part of the environments they are intended to model, creating feedback dynamics that affect future data collection, model retraining, and downstream decision-making.

Here, we address the broader ethical and systemic challenges associated with the deployment of machine learning technologies. It examines how feedback loops between models and environments can reinforce bias, how human-AI collaboration introduces new risks and responsibilities, and how conflicts between stakeholder values complicate the operationalization of fairness and accountability. In addition, it considers the role of contestability and institutional governance in sustaining responsible system behavior. These considerations highlights that responsibility is not a static property of an algorithm, but a dynamic outcome of system design, usage, and oversight over time.

16.6.1 System Feedback Loops

Machine learning systems do not merely observe and model the world—they also shape it. Once deployed, their predictions and decisions often influence the environments they are intended to analyze. This feedback alters future data distributions, modifies user behavior, and affects institutional practices, creating a recursive loop between model outputs and system inputs. Over time, such dynamics can amplify biases, entrench disparities, or unintentionally shift the objectives a model was designed to serve.

A well-documented example of this phenomenon is predictive policing[12]. When a model trained on historical arrest data predicts higher crime rates in a particular neighborhood, law enforcement may allocate more patrols to that area. This increased presence leads to more recorded incidents, which are then used as input for future model training—further reinforcing the models original prediction. Even if the model was not explicitly biased at the outset, its integration into a feedback loop results in a self-fulfilling pattern that disproportionately affects already over-policed communities.

Recommender systems[13] exhibit similar dynamics in digital environments. A content recommendation model that prioritizes engagement may gradually narrow the range of content a user is exposed to, leading to feedback loops that reinforce existing preferences or polarize opinions. These effects can be difficult to detect using conventional performance metrics, as the system continues to optimize its training objective even while diverging from broader social or epistemic goals.

[12] Use of historical crime data to predict future criminal activities and allocate police resources accordingly.

[13] Systems that help users discover products or content by predicting what's most relevant to their interests based on prior interactions.

From a systems perspective, feedback loops present a fundamental challenge to responsible AI. They undermine the assumption of independently and identically distributed data and complicate the evaluation of fairness, robustness, and generalization. Standard validation methods, which rely on static test sets, may fail to capture the evolving impact of the model on the data-generating process. Moreover, once such loops are established, interventions aimed at improving fairness or accuracy may have limited effect unless the underlying data dynamics are addressed.

Designing for responsibility in the presence of feedback loops requires a lifecycle view of machine learning systems. It entails not only monitoring model performance over time, but also understanding how the systems outputs influence the environment, how these changes are captured in new data, and how retraining practices either mitigate or exacerbate these effects.

In cloud-based systems, these updates may occur frequently and at scale, with extensive telemetry available to detect behavior drift. In contrast, edge and embedded deployments often operate offline or with limited observability. A smart home system that adapts thermostat behavior based on user interactions may reinforce energy consumption patterns or comfort preferences in ways that alter the home environment—and subsequently affect future inputs to the model. Without connectivity or centralized oversight, these loops may go unrecognized, despite their impact on both user behavior and system performance.

Systems must be equipped with mechanisms to detect distributional drift, identify behavior shaping effects, and support corrective updates that align with the systems intended goals. Feedback loops are not inherently harmful, but they must be recognized and managed. When left unexamined, they introduce systemic risk; when thoughtfully addressed, they provide an opportunity for learning systems to adapt responsibly in complex, dynamic environments.

16.6.2 Human-AI Collaboration and Oversight

Machine learning systems are increasingly deployed not as standalone agents, but as components in larger workflows that involve human decision-makers. In many domains, such as healthcare, finance, and transportation, models serve as decision-support tools, offering predictions, risk scores, or recommendations that are reviewed and acted upon by human operators. This collaborative configuration raises important questions about how responsibility is shared between humans and machines, how trust is calibrated, and how oversight mechanisms are implemented in practice.

Human-AI collaboration introduces both opportunities and risks. When designed appropriately, systems can augment human judgment, reduce cognitive burden, and enhance consistency in decision-making. However, when poorly designed, they may lead to automation bias, where users over-rely on model outputs even in the presence of clear errors. Conversely, excessive distrust can result in algorithm aversion, where users disregard useful model predictions due to a lack of transparency or perceived credibility. The effectiveness of collaborative systems depends not only on the model's performance, but on how the system communicates uncertainty, provides explanations, and allows for human override or correction.

Oversight mechanisms must be tailored to the deployment context. In high-stakes domains, such as medical triage or autonomous driving, humans may be expected to supervise automated decisions in real time. This configuration places cognitive and temporal demands on the human operator and assumes that intervention will occur quickly and reliably when needed. In practice, however, continuous human supervision is often impractical or ineffective, particularly when the operator must monitor multiple systems or lacks clear criteria for intervention.

From a systems design perspective, supporting effective oversight requires more than providing access to raw model outputs. Interfaces must be constructed to surface relevant information at the right time, in the right format, and with appropriate context. Confidence scores, uncertainty estimates, explanations, and change alerts can all play a role in enabling human oversight. Moreover, workflows must define when and how intervention is possible, who is authorized to override model outputs, and how such overrides are logged, audited, and incorporated into future system updates.

Consider a hospital triage system that uses a machine learning model to prioritize patients in the emergency department. The model generates a risk score for each incoming patient, which is presented alongside a suggested triage category. In principle, a human nurse is responsible for confirming or overriding the suggestion. However, if the model's outputs are presented without sufficient justification, such as an explanation of the contributing features or the context for uncertainty, the nurse may defer to the model even in borderline cases. Over time, the models outputs may become the de facto triage decision, especially under time pressure. If a distribution shift occurs (for instance, due to a new illness or change in patient demographics), the nurse may lack both the situational awareness and the interface support needed to detect that the model is underperforming. In such cases, the appearance of human oversight masks a system in which responsibility has effectively shifted to the model without clear accountability or recourse.

In such systems, human oversight is not merely a matter of policy declaration, but a function of infrastructure design: how predictions are surfaced, what information is retained, how intervention is enacted, and how feedback loops connect human decisions to system updates. Without integration across these components, oversight becomes fragmented, and responsibility may shift invisibly from human to machine.

The boundary between decision support and automation is often fluid. Systems initially designed to assist human decision-makers may gradually assume greater autonomy as trust increases or organizational incentives shift. This transition can occur without explicit policy changes, resulting in de facto automation without appropriate accountability structures. Responsible system design must therefore anticipate changes in use over time and ensure that appropriate checks remain in place even as reliance on automation grows.

Ultimately, human-AI collaboration requires careful integration of model capabilities, interface design, operational policy, and institutional oversight. Collaboration is not simply a matter of inserting a “human-in-the-loop”; it is a systems challenge that spans technical, organizational, and ethical dimensions. Designing for oversight entails embedding mechanisms that enable intervention, support informed trust, and support shared responsibility between human operators and machine learning systems.

16.6.3 Normative Pluralism and Value Conflicts

Responsible machine learning cannot be reduced to the optimization of a single objective. In real-world settings, machine learning systems are deployed into environments shaped by diverse, and often conflicting, human values. What constitutes a fair outcome for one stakeholder may be perceived as inequitable by another. Similarly, decisions that prioritize accuracy or efficiency may conflict with goals such as transparency, individual autonomy, or harm reduction. These tensions are not incidental—they are structural. They reflect the pluralistic nature of the societies in which machine learning systems are embedded and the institutional settings in which they are deployed.

Fairness is a particularly prominent site of value conflict. As discussed earlier in the chapter, fairness can be formalized in multiple, often incompatible ways. A model that satisfies demographic parity may violate equalized odds; a model that prioritizes individual fairness may undermine group-level parity[14]. Choosing among these definitions is not purely a technical decision but a normative one, informed by domain context, historical patterns of discrimination, and the perspectives of those affected by model outcomes. In practice, multiple stakeholders, including engineers, users, auditors, and regulators, may hold conflicting views on which definitions are most appropriate and why.

These tensions are not confined to fairness alone. Conflicts also arise between interpretability and predictive performance, privacy and personalization, or short-term utility and long-term consequences. These tradeoffs manifest differently depending on the systems deployment architecture, revealing how deeply value conflicts are tied to the design and operation of ML systems.

Consider a voice-based assistant deployed on a mobile device. To enhance personalization, the system may learn user preferences locally, without sending raw data to the cloud. This design improves privacy and reduces latency, but it may also lead to performance disparities if users with underrepresented usage patterns receive less accurate or responsive predictions. One way to improve fairness would be to centralize updates using group-level statistics—but doing so introduces new privacy risks and may violate user expectations around local data handling. Here, the design must navigate among valid but competing values: privacy, fairness, and personalization.

In cloud-based deployments, such as credit scoring platforms or recommendation engines, tensions often arise between transparency and proprietary protection. End users or regulators may demand clear explanations of why a decision was made, particularly in situations with significant consequences, but the models in use may rely on complex ensembles or proprietary training data. Revealing these internals may be commercially sensitive or technically infeasible. In such cases, the system must reconcile competing pressures for institutional accountability and business confidentiality.

[14] Group-level Parity: A requirement that different demographic groups receive similar benefits or outcomes.

In edge systems, such as home security cameras or autonomous drones, resource constraints often dictate model selection and update frequency. Prioritizing low latency and energy efficiency may require deploying compressed or quantized models that are less robust to distribution shift or adversarial perturbations. More resilient models could improve safety, but they may exceed the systems memory budget or violate power constraints. Here, safety, efficiency, and maintainability must be balanced under hardware-imposed tradeoffs.

On TinyML platforms, where models are deployed to microcontrollers with no persistent connectivity, tradeoffs are even more pronounced. A system may be optimized for static performance on a fixed dataset, but unable to incorporate new fairness constraints, retrain on updated inputs, or generate explanations once deployed. The value conflict lies not just in what the model optimizes, but in what the system is able to support post-deployment.

These examples make clear that normative pluralism[15] is not an abstract philosophical challenge; it is a recurring systems constraint. Technical approaches such as multi-objective optimization, constrained training[16], and fairness-aware evaluation can help surface and formalize tradeoffs, but they do not eliminate the need for judgment. Decisions about whose values to represent, which harms to mitigate, and how to balance competing objectives cannot be made algorithmically. They require deliberation, stakeholder input, and governance structures that extend beyond the model itself.

Participatory[17] and value-sensitive design[18] methodologies offer potential paths forward. Rather than treating values as parameters to be optimized after deployment, these approaches seek to engage stakeholders during the requirements phase, define ethical tradeoffs explicitly, and trace how they are instantiated in system architecture. While no design process can satisfy all values simultaneously, systems that are transparent about their tradeoffs and open to revision are better positioned to sustain trust and accountability over time.

Ultimately, machine learning systems are not neutral tools. They embed and enact value judgments, whether explicitly specified or implicitly assumed. A commitment to responsible AI requires acknowledging this fact and building systems that reflect and respond to the ethical and social pluralism of their operational contexts.

16.6.4 Transparency and Contestability

Transparency is widely recognized as a foundational principle of responsible machine learning. It enables users, developers, auditors, and regulators to understand how a system functions, assess its limitations, and identify sources of harm. Yet transparency alone is not sufficient. In high-stakes domains, individuals and institutions must not only understand system behavior—they must also be able to challenge, correct, or reverse it when necessary. This capacity for contestability, which refers to the ability to interrogate and contest a system's decisions, is a critical feature of accountability.

Transparency in machine learning systems typically focuses on disclosure: revealing how models are trained, what data they rely on, what assumptions are embedded in their design, and what known limitations affect their use. Doc-

[15] Normative pluralism refers to the presence of multiple, often conflicting, ethical frameworks within a society.

[16] Constrained Training: Training models under specific rules or constraints to satisfy fairness, privacy, or other criteria.

[17] Participatory Design: An approach that actively involves stakeholders in the design process to ensure the results meet their needs.

[18] Value-sensitive Design: A methodologically innovative approach that integrates ethical and moral considerations into the design process.

umentation tools such as model cards[19] and datasheets for datasets[20] support this goal by formalizing system metadata in a structured, reproducible format. These resources can improve governance, support compliance, and inform user expectations. However, transparency as disclosure does not guarantee meaningful control. Even when technical details are available, users may lack the institutional leverage, interface tools, or procedural access to contest a decision that adversely affects them.

To move from transparency to contestability, machine learning systems must be designed with mechanisms for explanation, recourse, and feedback. Explanation refers to the capacity of the system to provide understandable reasons for its outputs, tailored to the needs and context of the person receiving them. Recourse refers to the ability of individuals to alter their circumstances and receive a different outcome. Feedback refers to the ability of users to report errors, dispute outcomes, or signal concerns—and to have those signals incorporated into system updates or oversight processes.

These mechanisms are often lacking in practice, particularly in systems deployed at scale or embedded in low-resource devices. For example, in mobile loan application systems, users may receive a rejection without explanation and have no opportunity to provide additional information or appeal the decision. The lack of transparency at the interface level, even if documentation exists elsewhere, makes the system effectively unchallengeable. Similarly, a predictive model deployed in a clinical setting may generate a risk score that guides treatment decisions without surfacing the underlying reasoning to the physician. If the model underperforms for a specific patient subgroup, and this behavior is not observable or contestable, the result may be unintentional harm that cannot be easily diagnosed or corrected.

From a systems perspective, enabling contestability requires coordination across technical and institutional components. Models must expose sufficient information to support explanation. Interfaces must surface this information in a usable and timely way. Organizational processes must be in place to review feedback, respond to appeals, and update system behavior. Logging and auditing infrastructure must track not only model outputs, but user interventions and override decisions. In some cases, technical safeguards, including human-in-the-loop overrides and decision abstention thresholds, may also serve contestability by ensuring that ambiguous or high-risk decisions defer to human judgment.

The degree of contestability that is feasible varies by deployment context. In centralized cloud platforms, it may be possible to offer full explanation APIs, user dashboards, and appeal workflows. In contrast, in edge and TinyML deployments, contestability may be limited to logging and periodic updates based on batch-synchronized feedback. In all cases, the design of machine learning systems must acknowledge that transparency is not simply a matter of technical disclosure. It is a structural property of systems that determines whether users and institutions can meaningfully question, correct, and govern the behavior of automated decision-making.

[19] Model Cards: Tool that provides essential information about a machine learning model's capabilities and biases.

[20] Datasheets for Datasets: Documentation that describes a dataset's creation, composition, and intended use.

16.6.5 Institutional Embedding of Responsibility

Machine learning systems do not operate in isolation. Their development, deployment, and ongoing management are embedded within institutional environments that include technical teams, legal departments, product owners, compliance officers, and external stakeholders. Responsibility in such systems is not the property of a single actor or component—it is distributed across roles, workflows, and governance processes. Designing for responsible AI therefore requires attention to the institutional settings in which these systems are built and used.

This distributed nature of responsibility introduces both opportunities and challenges. On the one hand, the involvement of multiple stakeholders provides checks and balances that can help prevent harmful outcomes. On the other hand, the diffusion of responsibility can lead to accountability gaps, where no individual or team has clear authority or incentive to intervene when problems arise. When harm occurs, it may be unclear whether the fault lies with the data pipeline, the model architecture, the deployment configuration, the user interface, or the surrounding organizational context.

One illustrative case is Google Flu Trends[21], a widely cited example of failure due to institutional misalignment. The system, which attempted to predict flu outbreaks from search data, initially performed well but gradually diverged from reality due to changes in user behavior and shifts in the data distribution. These issues went uncorrected for years, in part because there were no established processes for system validation, external auditing, or escalation when model performance declined. The failure was not due to a single technical flaw, but to the absence of an institutional framework that could respond to drift, uncertainty, and feedback from outside the development team.

Embedding responsibility institutionally requires more than assigning accountability. It requires the design of processes, tools, and incentives that enable responsible action. Technical infrastructure such as versioned model registries, model cards, and audit logs must be coupled with organizational structures such as ethics review boards, model risk committees, and red-teaming procedures. These mechanisms ensure that technical insights are actionable, that feedback is integrated across teams, and that concerns raised by users, developers, or regulators are addressed systematically rather than ad hoc.

The level of institutional support required varies across deployment contexts. In large-scale cloud platforms, governance structures may include internal accountability audits, compliance workflows, and dedicated teams responsible for monitoring system behavior. In smaller-scale deployments, including edge or mobile systems embedded in healthcare devices or public infrastructure, governance may rely on cross-functional engineering practices and external certification or regulation. In TinyML deployments, where connectivity and observability are limited, institutional responsibility may be exercised through upstream controls such as safety-critical validation, embedded security constraints, and lifecycle tracking of deployed firmware.

In all cases, responsible machine learning requires coordination between technical and institutional systems. This coordination must extend across the entire model lifecycle—from initial data acquisition and model training to de-

[21] A project by Google to estimate flu activity using search engine queries, known for its public predictive inaccuracies.

ployment, monitoring, update, and eventual decommissioning. It must also incorporate external actors, including domain experts, civil society organizations, and regulatory authorities, to ensure that responsibility is exercised not only within the development team but across the broader ecosystem in which machine learning systems operate.

Responsibility is not a static attribute of a model or a team; it is a dynamic property of how systems are governed, maintained, and contested over time. Embedding that responsibility within institutions, by means of policy, infrastructure, and accountability mechanisms, is essential for aligning machine learning systems with the social values and operational realities they are meant to serve.

16.7 Implementation Challenges

While the principles and methods of responsible machine learning are increasingly well understood, their consistent implementation in real-world systems remains a significant challenge. Translating ethical intentions into sustained operational practice requires coordination across teams, infrastructure layers, data pipelines, and model lifecycle stages. In many cases, the barriers are not primarily technical, including the computation of fairness metrics or privacy guarantees, but organizational: unclear ownership, misaligned incentives, infrastructure limitations, or the absence of mechanisms to propagate responsibility across modular system components. Even when responsibility is treated as a design goal, it may be deprioritized during deployment, undercut by resource constraints, or rendered infeasible by limitations in data access, runtime support, or evaluation tooling.

This section examines the practical challenges that arise when embedding responsible AI practices into production ML systems. These include issues of organizational structure and accountability, limitations in data quality and availability, tensions between competing optimization objectives, breakdowns in lifecycle maintainability, and gaps in system-level evaluation. Collectively, these challenges illustrate the friction between idealized principles and operational reality—and highlight the importance of systems-level strategies that embed responsibility into the architecture, infrastructure, and workflows of machine learning deployment.

16.7.1 Organizational Structures and Incentives

The implementation of responsible machine learning is shaped not only by technical feasibility but by the organizational context in which systems are developed and deployed. Within companies, research labs, and public institutions, responsibility must be translated into concrete roles, workflows, and incentives. In practice, however, organizational structures often fragment responsibility, making it difficult to coordinate ethical objectives across engineering, product, legal, and operational teams.

Responsible AI requires sustained investment in practices such as subgroup performance evaluation, explainability analysis, adversarial robustness testing, and the integration of privacy-preserving techniques like differential privacy or federated training. These activities can be time-consuming and resource-intensive, yet they often fall outside the formal performance metrics used to evaluate team productivity. For example, teams may be incentivized to ship features quickly or meet performance benchmarks, even when doing so undermines fairness or overlooks potential harms. When ethical diligence is treated as a discretionary task, instead of being an integrated component of the system lifecycle, it becomes vulnerable to deprioritization under deadline pressure or organizational churn.

Responsibility is further complicated by ambiguity over ownership. In many organizations, no single team is responsible for ensuring that a system behaves ethically over time. Model performance may be owned by one team, user experience by another, data infrastructure by a third, and compliance by a fourth. When issues arise, including disparate impact in predictions or insufficient explanation quality, there may be no clear protocol for identifying root causes or coordinating mitigation. As a result, concerns raised by developers, users, or auditors may go unaddressed, not because of malicious intent, but due to lack of process and cross-functional alignment.

Establishing effective organizational structures for responsible AI requires more than policy declarations. It demands operational mechanisms: designated roles with responsibility for ethical oversight, clearly defined escalation pathways, accountability for post-deployment monitoring, and incentives that reward teams for ethical foresight and system maintainability. In some organizations, this may take the form of Responsible AI committees, cross-functional review boards, or model risk teams that work alongside developers throughout the model lifecycle. In others, domain experts or user advocates may be embedded into product teams to anticipate downstream impacts and evaluate value tradeoffs in context.

As shown in Figure 16.8, the responsibility for ethical system behavior is distributed across multiple constituencies, including industry, academia, civil society, and government. Within organizations, this distribution must be mirrored by mechanisms that connect technical design with strategic oversight and operational control. Without these linkages, responsibility becomes diffuse, and well-intentioned efforts may be undermined by systemic misalignment.

Ultimately, responsible AI is not merely a question of technical excellence or regulatory compliance. It is a systems-level challenge that requires aligning ethical objectives with the institutional structures through which machine learning systems are designed, deployed, and maintained. Creating and sustaining these structures is essential for ensuring that responsibility is embedded not only in the model, but in the organization that governs its use.

16.7.2 Data Constraints and Quality Gaps

Despite broad recognition that data quality is essential for responsible machine learning, improving data pipelines remains one of the most difficult implementation challenges in practice. Developers and researchers often understand the importance of representative data, accurate labeling, and mitigation of historical bias. Yet even when intentions are clear, structural and organizational barriers frequently prevent meaningful intervention. Responsibility for data is often

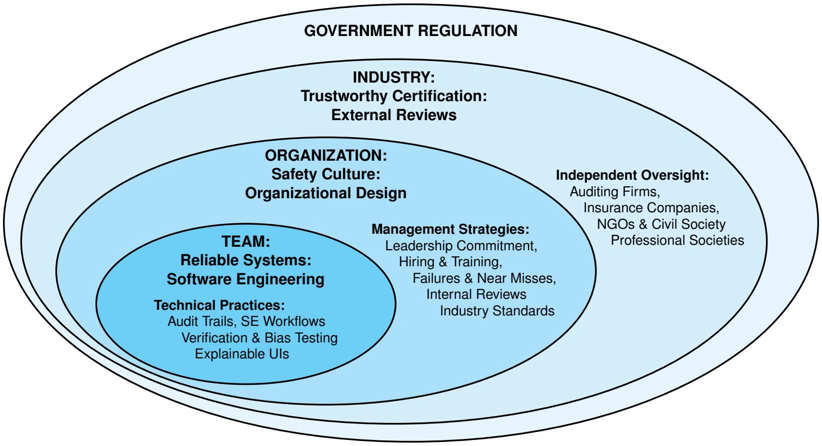


Figure 16.8: How various groups impact human-centered AI. Source: Shneiderman (2020).

distributed across teams, governed by legacy systems, or embedded in broader institutional processes that are difficult to change.

Subgroup imbalance[22], label ambiguity[23], and distribution shift, each of which affect generalization and performance across domains, are well-established concerns in responsible ML. These issues often manifest in the form of poor calibration, out-of-distribution failures, or demographic disparities in evaluation metrics. However, addressing them in real-world settings requires more than technical knowledge. It requires access to relevant data, institutional support for remediation, and sufficient time and resources to iterate on the dataset itself. In many machine learning pipelines, once the data is collected and the training set defined, the data pipeline becomes effectively frozen. Teams may lack both the authority and the infrastructure to modify or extend the dataset midstream, especially when data versioning and lineage tracking are tightly integrated into production analytics workflows.

However, addressing them in real-world settings requires more than technical knowledge. It requires access to relevant data, institutional support for remediation, and sufficient time and resources to iterate on the dataset itself. In many machine learning pipelines, once the data is collected and the training set defined, the data pipeline becomes effectively frozen. Teams may lack both the authority and the infrastructure to modify or extend the dataset midstream, even if performance disparities are discovered. Even in modern data pipelines with automated validation and feature stores, retroactively correcting training distributions remains difficult once dataset versioning and data lineage have been locked into production.

In domains like healthcare, education, and social services, these challenges are especially pronounced. Data acquisition may be subject to legal constraints, privacy regulations, or cross-organizational coordination. For example, a team developing a triage model may discover that their training data underrepresents patients from smaller or rural hospitals. Correcting this imbalance would require negotiating data access with external partners, aligning on feature standards, and resolving inconsistencies in labeling practices. Even when all

[22] Subgroup Imbalance: Refers to the uneven representation of various groups within a dataset, which can lead to biased machine learning models.

[23] Label Ambiguity: Occurs when labels assigned to data are unclear or inconsistently applied, complicating machine learning training and evaluation.

parties agree on the need for improvement, the logistical and operational costs can be prohibitive.

Efforts to collect more representative data may also run into ethical and political concerns. In some cases, additional data collection could expose marginalized populations to new risks. This paradox of exposure, in which the individuals most harmed by exclusion are also those most vulnerable to misuse, complicates efforts to improve fairness through dataset expansion. For example, gathering more data on non-binary individuals to support fairness in gender-sensitive applications may improve model coverage, but it also raises serious concerns around consent, identifiability, and downstream use. Teams must navigate these tensions carefully, often without clear institutional guidance.

Even when data is plentiful, upstream biases in data collection systems can persist unchecked. Many organizations rely on third-party data vendors, external APIs, or operational databases that were not designed with fairness or interpretability in mind. For instance, Electronic Health Records[24], which are commonly used in clinical machine learning, often reflect systemic disparities in care, as well as documentation habits that encode racial or socioeconomic bias (Himmelstein, Bates, and Zhou 2022). Teams working downstream may have little visibility into how these records were created, and few levers for addressing embedded harms.

Improving dataset quality is often not the responsibility of any one team. Data pipelines may be maintained by infrastructure or analytics groups that operate independently of the ML engineering or model evaluation teams. This organizational fragmentation makes it difficult to coordinate data audits, track provenance, or implement feedback loops that connect model behavior to underlying data issues. In practice, responsibility for dataset quality tends to fall through the cracks—recognized as important, but rarely prioritized or resourced.

Addressing these challenges requires long-term investment in infrastructure, workflows, and cross-functional communication. Technical tools such as data validation, automated audits, and dataset documentation frameworks (e.g., model cards, datasheets, or the Data Nutrition Project) can help, but only when they are embedded within teams that have the mandate and support to act on their findings. Ultimately, improving data quality is not just a matter of better tooling—it is a question of how responsibility for data is assigned, shared, and sustained across the system lifecycle.

16.7.3 Balancing Competing Objectives

Machine learning system design is often framed as a process of optimization—improving accuracy, reducing loss, or maximizing utility. Yet in responsible ML practice, optimization must be balanced against a range of competing objectives, including fairness, interpretability, robustness, privacy, and resource efficiency. These objectives are not always aligned, and improvements in one dimension may entail tradeoffs in another. While these tensions are well understood in theory, managing them in real-world systems is a persistent and unresolved challenge.

[24] Electronic Health Records (EHR): Digital versions of patients' medical histories, used extensively in healthcare for data analysis and predictive modeling.

Consider the tradeoff between model accuracy and interpretability. In many cases, more interpretable models, including shallow decision trees and linear models, achieve lower predictive performance than complex ensemble methods or deep neural networks. In low-stakes applications, this tradeoff may be acceptable, or even preferred. But in high-stakes domains such as healthcare or finance, where decisions affect individuals well-being or access to opportunity, teams are often caught between the demand for performance and the need for transparent reasoning. Even when interpretability is prioritized during development, it may be overridden at deployment in favor of marginal gains in model accuracy.

Similar tensions emerge between personalization and fairness. A recommendation system trained to maximize user engagement may personalize aggressively, using fine-grained behavioral data to tailor outputs to individual users. While this approach can improve satisfaction for some users, it may entrench disparities across demographic groups, particularly if personalization draws on features correlated with race, gender, or socioeconomic status. Adding fairness constraints may reduce disparities at the group level, but at the cost of reducing perceived personalization for some users. These effects are often difficult to measure, and even more difficult to explain to product teams under pressure to optimize engagement metrics.

Privacy introduces another set of constraints. Techniques such as differential privacy, federated learning, or local data minimization can meaningfully reduce privacy risks. But they also introduce noise, limit model capacity, or reduce access to training data. In centralized systems, these costs may be absorbed through infrastructure scaling or hybrid training architectures. In edge or TinyML deployments, however, the tradeoffs are more acute. A wearable device tasked with local inference must often balance model complexity, energy consumption, latency, and privacy guarantees simultaneously. Supporting one constraint typically weakens another, forcing system designers to prioritize among equally important goals. These tensions are further amplified by deployment-specific design decisions such as quantization levels, activation clipping, or compression strategies that affect how effectively models can support multiple objectives at once.

These tradeoffs are not purely technical—they reflect deeper normative judgments about what a system is designed to achieve and for whom. Responsible ML development requires making these judgments explicit, evaluating them in context, and subjecting them to stakeholder input and institutional oversight. Multi-objective optimization frameworks can formalize some of these tradeoffs mathematically, but they cannot resolve value conflicts or prescribe the "right" balance. In many cases, tradeoffs are revisited multiple times over a systems lifecycle, as deployment conditions change, metrics evolve, or stakeholder expectations shift. Designing for constraint-aware tradeoffs may leverage techniques such as Pareto optimization[25] or parameter-efficient fine-tuning, but value tradeoffs must still be surfaced, discussed, and governed explicitly.

What makes this challenge particularly difficult in implementation is that these competing objectives are rarely owned by a single team or function. Performance may be optimized by the modeling team, fairness monitored by a responsible AI group, and privacy handled by legal or compliance departments.

[25] Pareto Optimization: A method in decision-making that searches for solutions where no objective can be improved without worsening others.

Without deliberate coordination, system-level tradeoffs can be made implicitly, piecemeal, or without visibility into long-term consequences. Over time, the result may be a model that appears well-behaved in isolation but fails to meet its ethical goals when embedded in production infrastructure.

Balancing competing objectives requires not only technical fluency but a commitment to transparency, deliberation, and alignment across teams. Systems must be designed to surface tradeoffs rather than obscure them, to make room for constraint-aware development rather than pursue narrow optimization. In practice, this may require redefining what "success" looks like—not as performance on a single metric, but as sustained alignment between system behavior and its intended role in a broader social or operational context.

16.7.4 Scalability and Maintenance

Responsible machine learning practices are often introduced during the early phases of model development: fairness audits are conducted during initial evaluation, interpretability methods are applied during model selection, and privacy-preserving techniques are considered during training. However, as systems transition from research prototypes to production deployments, these practices frequently degrade or disappear. The gap between what is possible in principle and what is sustainable in production is a core implementation challenge for responsible AI.

Many responsible AI interventions are not designed with scalability in mind. Fairness checks may be performed on a static dataset, but not integrated into ongoing data ingestion pipelines. Explanation methods may be developed using development-time tools but never translated into deployable user-facing interfaces. Privacy constraints may be enforced during training, but overlooked during post-deployment monitoring or model updates. In each case, what begins as a responsible design intention fails to persist across system scaling and lifecycle changes.

Production environments introduce new pressures that reshape system priorities. Models must operate across diverse hardware configurations, interface with evolving APIs, serve millions of users with low latency, and maintain availability under operational stress. For instance, maintaining consistent behavior across CPU, GPU, and edge accelerators requires tight integration between framework abstractions, runtime schedulers, and hardware-specific compilers. These constraints demand continuous adaptation and rapid iteration, often deprioritizing activities that are difficult to automate or measure. Responsible AI practices, especially those that involve human review, stakeholder consultation, or post-hoc evaluation, may not be easily incorporated into fast-paced DevOps pipelines[26]. As a result, ethical commitments that are present at the prototype stage may be sidelined as systems mature.

Maintenance introduces further complexity. Machine learning systems are rarely static. New data is ingested, retraining is performed, features are deprecated or added, and usage patterns shift over time. In the absence of rigorous version control, changelogs, and impact assessments, it can be difficult to trace how system behavior evolves or whether responsibility-related properties such as fairness or robustness are being preserved. Moreover, organizational

[26] DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) aimed at shortening the systems development lifecycle.

turnover and team restructuring can erode institutional memory. Teams responsible for maintaining a deployed model may not be the ones who originally developed or audited it, leading to unintentional misalignment between system goals and current implementation. These issues are especially acute in continual or streaming learning scenarios, where concept drift[27] and shifting data distributions demand active monitoring and real-time updates.

These challenges are magnified in multi-model systems and cross-platform deployments. A recommendation engine may consist of dozens of interacting models, each optimized for a different subtask or user segment. A voice assistant deployed across mobile and edge environments may maintain different versions of the same model, tuned to local hardware constraints. Coordinating updates, ensuring consistency, and sustaining responsible behavior in such distributed systems requires infrastructure that tracks not only code and data, but also values and constraints.

Addressing scalability and maintenance challenges requires treating responsible AI as a lifecycle property, not a one-time evaluation. This means embedding audit hooks, metadata tracking, and monitoring protocols into system infrastructure. It also means creating documentation that persists across team transitions, defining accountability structures that survive project handoffs, and ensuring that system updates do not inadvertently erase hard-won improvements in fairness, transparency, or safety. While such practices can be difficult to implement retroactively, they can be integrated into system design from the outset through responsible-by-default tooling and workflows.

Ultimately, responsibility must scale with the system. Machine learning models deployed in real-world environments must not only meet ethical standards at launch, but continue to do so as they grow in complexity, user reach, and operational scope. Achieving this requires sustained organizational investment and architectural planning—not simply technical correctness at a single point in time.

16.7.5 Standardization and Evaluation Gaps

While the field of responsible machine learning has produced a wide range of tools, metrics, and evaluation frameworks, there is still little consensus on how to systematically assess whether a system is responsible in practice. Many teams recognize the importance of fairness, privacy, interpretability, and robustness, yet they often struggle to translate these principles into consistent, measurable standards. The lack of formalized evaluation criteria, combined with the fragmentation of tools and frameworks, poses a significant barrier to implementing responsible AI at scale.

This fragmentation is evident both across and within institutions. Academic research frequently introduces new metrics for fairness or robustness that are difficult to reproduce outside experimental settings. Industrial teams, by contrast, must prioritize metrics that integrate cleanly with production infrastructure, are interpretable by non-specialists, and can be monitored over time. As a result, practices developed in one context may not transfer well to another, and performance comparisons across systems may be unreliable or misleading. For instance, a model evaluated for fairness on one benchmark dataset using de-

[27] Concept drift occurs when the statistical properties of the target variable change over time in unforeseen ways.

mographic parity may not meet the requirements of equalized odds in another domain or jurisdiction. Without shared standards, these evaluations remain ad hoc, making it difficult to establish confidence in a systems responsible behavior across contexts.

Responsible AI evaluation also suffers from a mismatch between the unit of analysis, which is frequently the individual model or batch job, and the level of deployment, which includes end-to-end system components such as data ingestion pipelines, feature transformations, inference APIs, caching layers, and human-in-the-loop workflows. A system that appears fair or interpretable in isolation may fail to uphold those properties once integrated into a broader application. Tools that support holistic, system-level evaluation remain underdeveloped, and there is little guidance on how to assess responsibility across interacting components in modern ML stacks.

Further complicating matters is the lack of lifecycle-aware metrics. Most evaluation tools are applied at a single point in time—often just before deployment. Yet responsible AI properties such as fairness and robustness are dynamic. They depend on how data distributions evolve, how models are updated, and how users interact with the system. Without continuous or periodic evaluation, it is difficult to determine whether a system remains aligned with its intended ethical goals after deployment. Post-deployment monitoring tools exist, but they are rarely integrated with the development-time metrics used to assess initial model quality. This disconnect makes it hard to detect drift in ethical performance, or to trace observed harms back to their upstream sources.

Tool fragmentation further contributes to these challenges. Responsible AI tooling is often distributed across disconnected packages, dashboards, or internal systems, each designed for a specific task or metric. A team may use one tool for explainability, another for bias detection, and a third for compliance reporting—with no unified interface for reasoning about system-level tradeoffs. The lack of interoperability hinders collaboration between teams, complicates documentation, and increases the risk that important evaluations will be skipped or performed inconsistently. These challenges are compounded by missing hooks for metadata propagation or event logging across components like feature stores, inference gateways, and model registries.

Addressing these gaps requires progress on multiple fronts. First, shared evaluation frameworks must be developed that define what it means for a system to behave responsibly—not just in abstract terms, but in measurable, auditable criteria that are meaningful across domains. Second, evaluation must be extended beyond individual models to cover full system pipelines, including user-facing interfaces, update policies, and feedback mechanisms. Finally, evaluation must become a recurring lifecycle activity, supported by infrastructure that tracks system behavior over time and alerts developers when ethical properties degrade.

Without standardized, system-aware evaluation methods, responsible AI remains a moving target—described in principles but difficult to verify in practice. Building confidence in machine learning systems requires not only better models and tools, but shared norms, durable metrics, and evaluation practices that reflect the operational realities of deployed AI.

Responsible AI cannot be achieved through isolated interventions or static compliance checks. It requires architectural planning, infrastructure support, and institutional processes that sustain ethical goals across the system lifecycle. As ML systems scale, diversify, and embed themselves into sensitive domains, the ability to enforce properties like fairness, robustness, and privacy must be supported not only at model selection time, but across retraining, quantization, serving, and monitoring stages. Without persistent oversight, responsible practices degrade as systems evolve—especially when tooling, metrics, and documentation are not designed to track and preserve them through deployment and beyond.

Meeting this challenge will require greater standardization, deeper integration of responsibility-aware practices into CI/CD pipelines, and long-term investment in system infrastructure that supports ethical foresight. The goal is not to perfect ethical decision-making in code, but to make responsibility an operational property—traceable, testable, and aligned with the constraints and affordances of machine learning systems at scale.

16.8 AI Safety and Value Alignment

While earlier sections focused on robustness and safety as technical properties, such as resisting distribution shift or adversarial inputs, AI safety[28] in a broader sense concerns the behavior of increasingly autonomous systems that may act in ways misaligned with human goals. Beyond the robustness of individual models, the field of AI safety examines how to ensure that machine learning systems optimize for the right objectives and remain under meaningful human control.

As machine learning systems increase in autonomy, scale, and deployment complexity, the nature of responsibility expands beyond model-level fairness or privacy concerns. It includes ensuring that systems pursue the right objectives, behave safely in uncertain environments, and remain aligned with human intentions over time. These concerns fall under the domain of AI safety, which focuses on preventing unintended or harmful outcomes from capable AI systems. A central challenge is that today's ML models often optimize proxy metrics, such as loss functions, reward functions, or engagement signals, that do not fully capture human values.

One concrete example comes from recommendation systems, where a model trained to maximize click-through rate (CTR) may end up promoting content that increases engagement but diminishes user satisfaction, including clickbait, misinformation, and emotionally manipulative material[29]. This behavior is aligned with the proxy, but misaligned with the actual goal, resulting in a feedback loop that reinforces undesirable outcomes. As shown in Figure 16.9, the system learns to optimize for a measurable reward (clicks) rather than the intended human-centered outcome (satisfaction). The result is emergent behavior that reflects specification gaming or reward hacking—a central concern in value alignment and AI safety.

In 1960, Norbert Wiener wrote, "if we use, to achieve our purposes, a mechanical agency with whose operation we cannot interfere effectively… we had

[28] Some practitioners prefer the term "AI reliability" over "AI safety," as it emphasizes the positive goal of consistent, dependable performance rather than focusing on risk mitigation. Both terms refer to similar underlying principles of ensuring ML systems behave as intended.

[29] Reward hacking refers to strategies where systems optimize for observable metrics that do not align with actual objectives, leading often to unwanted behaviors.

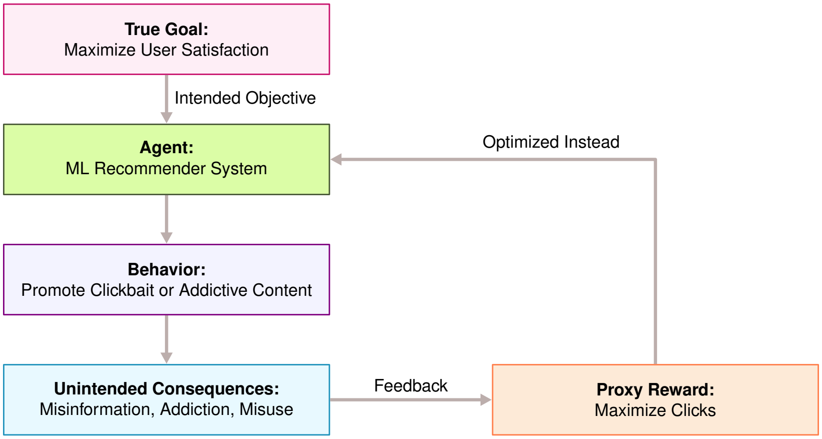


Figure 16.9: Misalignment between a recommender system's true objective and its optimized reward function. The model optimizes for click-through rate (a proxy for satisfaction), leading to unintended behaviors such as clickbait and misinformation. These behaviors are reinforced through feedback, illustrating the problem of reward hacking and the challenge of aligning ML systems with human values.

better be quite sure that the purpose put into the machine is the purpose which we desire" (Wiener 1960).

As the capabilities of deep learning models have increasingly approached, and, in certain instances, exceeded, human performance, the concern that such systems may pursue unintended or undesirable goals has become more pressing (S. Russell 2021). Within the field of AI safety, a central focus is the problem of value alignment: how to ensure that machine learning systems act in accordance with broad human intentions, rather than optimizing misaligned proxies or exhibiting emergent behavior that undermines social goals. As Russell argues in Human-Compatible Artificial Intelligence, much of current AI research presumes that the objectives to be optimized are known and fixed, focusing instead on the effectiveness of optimization rather than the design of objectives themselves.

Yet defining "the right purpose" for intelligent systems is especially difficult in real-world deployment settings. ML systems often operate within dynamic environments, interact with multiple stakeholders, and adapt over time. These conditions make it challenging to encode human values in static objective functions or reward signals. Frameworks like Value Sensitive Design aim to address this challenge by providing formal processes for eliciting and integrating stakeholder values during system design.

Taking a holistic sociotechnical perspective, which accounts for both the algorithmic mechanisms and the contexts in which systems operate, is essential for ensuring alignment. Without this, intelligent systems may pursue narrow performance objectives (e.g., accuracy, engagement, or throughput) while producing socially undesirable outcomes. Achieving robust alignment under such conditions remains an open and critical area of research in ML systems.

The absence of alignment can give rise to well-documented failure modes, particularly in systems that optimize complex objectives. In reinforcement learning (RL), for example, models often learn to exploit unintended aspects of the reward function—a phenomenon known as specification gaming or reward hacking. Such failures arise when variables not explicitly included in the objective are manipulated in ways that maximize reward while violating human intent.

A particularly influential approach in recent years has been reinforcement learning from human feedback (RLHF), where large pre-trained models are fine-tuned using human-provided preference signals (Christiano et al. 2017). While this method improves alignment over standard RL, it also introduces new risks. Ngo (Ngo, Chan, and Mindermann 2022) identifies three potential failure modes introduced by RLHF: (1) situationally aware reward hacking, where models exploit human fallibility; (2) the emergence of misaligned internal goals that generalize beyond the training distribution; and (3) the development of power-seeking behavior that preserves reward maximization capacity, even at the expense of human oversight.

These concerns are not limited to speculative scenarios. Amodei et al. (2016) outline six concrete challenges for AI safety: (1) avoiding negative side effects during policy execution, (2) mitigating reward hacking, (3) ensuring scalable oversight when ground-truth evaluation is expensive or infeasible, (4) designing safe exploration strategies that promote creativity without increasing risk, (5) achieving robustness to distributional shift in testing environments, and (6) maintaining alignment across task generalization. Each of these challenges becomes more acute as systems are scaled up, deployed across diverse settings, and integrated with real-time feedback or continual learning.

16.8.1 Autonomous Systems and Trust

The consequences of autonomous systems that act independently of human oversight and often outside the bounds of human judgment have been widely documented across multiple industries. A prominent recent example is the suspension of Cruises deployment and testing permits by the California Department of Motor Vehicles due to "unreasonable risks to public safety". One such incident involved a pedestrian who entered a crosswalk just as the stoplight turned green—an edge case in perception and decision-making that led to a collision. A more tragic example occurred in 2018, when a self-driving Uber vehicle in autonomous mode failed to classify a pedestrian pushing a bicycle as an object requiring avoidance, resulting in a fatality.

While autonomous driving systems are often the focal point of public concern, similar risks arise in other domains. Remotely piloted drones and autonomous military systems are already reshaping modern warfare, raising not only safety and effectiveness concerns but also difficult questions about ethical oversight, rules of engagement, and responsibility. When autonomous systems fail, the question of who should be held accountable remains both legally and ethically unresolved.

At its core, this challenge reflects a deeper tension between human and machine autonomy. Engineering and computer science disciplines have historically emphasized machine autonomy—improving system performance, minimizing human intervention, and maximizing automation. A bibliometric analysis of the ACM Digital Library found that, as of 2019, 90% of the most cited papers referencing "autonomy" focused on machine, rather than human, autonomy (Calvo et al. 2020). Productivity, efficiency, and automation have been widely treated as default objectives, often without interrogating the assumptions or tradeoffs they entail for human agency and oversight.

However, these goals can place human interests at risk when systems operate in dynamic, uncertain environments where full specification of safe behavior is infeasible. This difficulty is formally captured by the frame problem[30] and qualification problem[31], both of which highlight the impossibility of enumerating all the preconditions and contingencies needed for real-world action to succeed (McCarthy 1981). In practice, such limitations manifest as brittle autonomy: systems that appear competent under nominal conditions but fail silently or dangerously when faced with ambiguity or distributional shift.

To address this, researchers have proposed formal safety frameworks such as Responsibility-Sensitive Safety (RSS) (Shalev-Shwartz, Shammah, and Shashua 2017), which decompose abstract safety goals into mathematically defined constraints on system behavior—such as minimum distances, braking profiles, and right-of-way conditions. These formulations allow safety properties to be verified under specific assumptions and scenarios. However, such approaches remain vulnerable to the same limitations they aim to solve: they are only as good as the assumptions encoded into them and often require extensive domain modeling that may not generalize well to unanticipated edge cases.

An alternative approach emphasizes human-centered system design, ensuring that human judgment and oversight remain central to autonomous decision-making. Value-Sensitive Design (Friedman 1996) proposes incorporating user values into system design by explicitly considering factors like capability, complexity, misrepresentation, and the fluidity of user control. More recently, the METUX model (Motivation, Engagement, and Thriving in the User Experience) extends this thinking by identifying six “spheres of technology experience”—Adoption, Interface, Tasks, Behavior, Life, and Society, which affect how technology supports or undermines human flourishing (Peters, Calvo, and Ryan 2018). These ideas are rooted in Self-Determination Theory (SDT), which defines autonomy not as control in a technical sense, but as the ability to act in accordance with ones values and goals (Ryan and Deci 2000).

In the context of ML systems, these perspectives underscore the importance of designing architectures, interfaces, and feedback mechanisms that preserve human agency. For instance, recommender systems that optimize engagement metrics may interfere with behavioral autonomy by shaping user preferences in opaque ways. By evaluating systems across METUXs six spheres, designers can anticipate and mitigate downstream effects that compromise meaningful autonomy, even in cases where short-term system performance appears optimal.

16.8.2 AIs Economic Impact

A recurring concern in the adoption of AI technologies is the potential for widespread job displacement. As machine learning systems become capable of performing increasingly complex cognitive and physical tasks, there is growing fear that they may replace existing workers and reduce the availability of alternative employment opportunities across industries. These concerns are particularly acute in sectors with well-structured tasks, including logistics,

[30] Frame Problem: Difficulty in specifying all the necessary operational conditions beforehand for automated agents.

[31] Qualification Problem: Challenge in artificially intelligent systems to acknowledge exceptional circumstances.

manufacturing, and customer service, where AI-based automation appears both technically feasible and economically incentivized.

However, the economic implications of automation are not historically unprecedented. Prior waves of technological change, including industrial mechanization and computerization, have tended to result in job displacement rather than absolute job loss (Shneiderman 2022). Automation often reduces the cost and increases the quality of goods and services, thereby expanding access and driving demand. This demand, in turn, creates new forms of production, distribution, and support work—sometimes in adjacent sectors, sometimes in roles that did not previously exist.

Empirical studies of industrial robotics and process automation further challenge the feasibility of "lights-out" factories, systems that are designed for fully autonomous operation without human oversight. Despite decades of effort, most attempts to achieve this level of automation have been unsuccessful. According to the MIT Work of the Future task force, such efforts often lead to zero-sum automation, where productivity increases come at the expense of system flexibility, adaptability, and fault tolerance. Human workers remain essential for tasks that require contextual judgment, cross-domain generalization, or system-level debugging—capabilities that are still difficult to encode in machine learning models or automation frameworks.

Instead, the task force advocates for a positive-sum automation approach that augments human work rather than replacing it. This strategy emphasizes the integration of AI systems into workflows where humans retain oversight and control, such as semi-autonomous assembly lines or collaborative robotics. It also recommends bottom-up identification of automatable tasks, with priority given to those that reduce cognitive load or eliminate hazardous work, alongside the selection of appropriate metrics that capture both efficiency and resilience. Metrics rooted solely in throughput or cost minimization may inadvertently penalize human-in-the-loop designs, whereas broader metrics tied to safety, maintainability, and long-term adaptability provide a more comprehensive view of system performance.

Nonetheless, the long-run economic trajectory does not eliminate the reality of near-term disruption. Workers whose skills are rendered obsolete by automation may face wage stagnation, reduced bargaining power, or long-term displacement—especially in the absence of retraining opportunities or labor market mobility. Public and legislative efforts will play a critical role in shaping this transition, including policies that promote equitable access to the benefits of automation. These may include upskilling initiatives, social safety nets, minimum wage increases, and corporate accountability frameworks that ensure the distributional impacts of AI are monitored and addressed over time.

16.8.3 AI Literacy and Communication

A 1993 survey of 3,000 North American adults' beliefs about the "electronic thinking machine" revealed two dominant perspectives on early computing: the "beneficial tool of man" and the "awesome thinking machine" (Martin 1993). The latter reflects a perception of computers as mysterious, intelligent, and potentially uncontrollable—"smarter than people, unlimited, fast, and frightening." These perceptions, though decades old, remain relevant in the age of machine learning systems. As the pace of innovation accelerates, responsible AI development must be accompanied by clear and accurate scientific communication, especially concerning the capabilities, limitations, and uncertainties of AI technologies.

As modern AI systems surpass layperson understanding and begin to influence high-stakes decisions, public narratives tend to polarize between utopian and dystopian extremes. This is not merely a result of media framing, but of a more fundamental difficulty: in technologically advanced societies, the outputs of scientific systems are often perceived as magical—"understandable only in terms of what it did, not how it worked" (Handlin 1965). Without scaffolding for technical comprehension, systems like generative models, autonomous agents, or large-scale recommender platforms can be misunderstood or mistrusted, impeding informed public discourse.

Tech companies bear responsibility in this landscape. Overstated claims, anthropomorphic marketing, or opaque product launches contribute to cycles of hype and disappointment, eroding public trust. But improving AI literacy requires more than restraint in corporate messaging. It demands systematic research on scientific communication in the context of AI. Despite the societal impact of modern machine learning, an analysis of the Scopus scholarly database found only a small number of papers that intersect the domains of "artificial intelligence" and "science communication" (Schäfer 2023).

Addressing this gap requires attention to how narratives about AI are shaped—not just by companies, but also by academic institutions, regulators, journalists, non-profits, and policy advocates. The frames and metaphors used by these actors significantly influence how the public perceives agency, risk, and control in AI systems (Lindgren 2023). These perceptions, in turn, affect adoption, oversight, and resistance, particularly in domains such as education, healthcare, and employment, where AI deployment intersects directly with lived experience.

From a systems perspective, public understanding is not an externality—it is part of the deployment context. Misinformation about how AI systems function can lead to overreliance, misplaced blame, or underutilization of safety mechanisms. Equally, a lack of understanding of model uncertainty, data bias, or decision boundaries can exacerbate the risks of automation-induced harm. For individuals whose jobs are impacted by AI, targeted efforts to build domain-specific literacy can also support reskilling and adaptation (Ng et al. 2021).

Ultimately, AI literacy is not just about technical fluency. It is about building public confidence that the goals of system designers are aligned with societal welfare—and that those building AI systems are not removed from public values, but accountable to them. As Handlin observed in 1965: *"Even those who never acquire that understanding need assurance that there is a connection between the goals of science and their welfare, and above all, that the scientist is not a man altogether apart but one who shares some of their value."*

16.9 Conclusion

Responsible artificial intelligence is essential as machine learning systems increasingly shape decisions in healthcare, employment, finance, and the justice system. While these technologies offer substantial benefits, their deployment without ethical safeguards risks amplifying harm—through biased predictions, opaque decision-making, privacy violations, and misaligned objectives.

To mitigate these risks, the principles of fairness, explainability, accountability, safety, and transparency must be operationalized throughout the ML lifecycle. Yet implementing these principles presents persistent challenges: detecting and correcting data imbalance, balancing predictive performance against interpretability or robustness, ensuring privacy in both centralized and edge settings, and maintaining alignment in systems that evolve after deployment. Frameworks such as value-sensitive design provide structured approaches for surfacing stakeholder values and navigating tradeoffs across competing system objectives.

Achieving responsible AI in practice will require sustained research, standardization, and institutional commitment. Robust benchmarks and evaluation frameworks are needed to compare model behavior under real-world constraints, particularly in terms of subgroup performance, distributional robustness, and privacy guarantees. As deployment extends to edge environments and personalized settings, new methods will be required to support lightweight explainability and user control in TinyML systems. Policy interventions and incentive structures must also be updated to prioritize long-term system reliability and ethical oversight over short-term performance gains.

Crucially, responsible AI is not reducible to technical metrics or checklists. It demands interdisciplinary collaboration, human-centered design, and continuous reflection on the social contexts in which systems operate. By embedding ethical considerations into infrastructure, workflows, and governance structures, the machine learning community can help ensure that AI serves broad societal interests. The challenge ahead lies in transforming ethical responsibility from an aspiration into a durable property of ML systems—and doing so at scale.

16.10 Resources

ℹ Slides

- What am I building? What is the goal?
- Who is the audience?
- What are the consequences?
- Responsible Data Collection.

❗ Videos

- *Coming soon.*

Exercises

- *Coming soon.*

Chapter 17

Sustainable AI

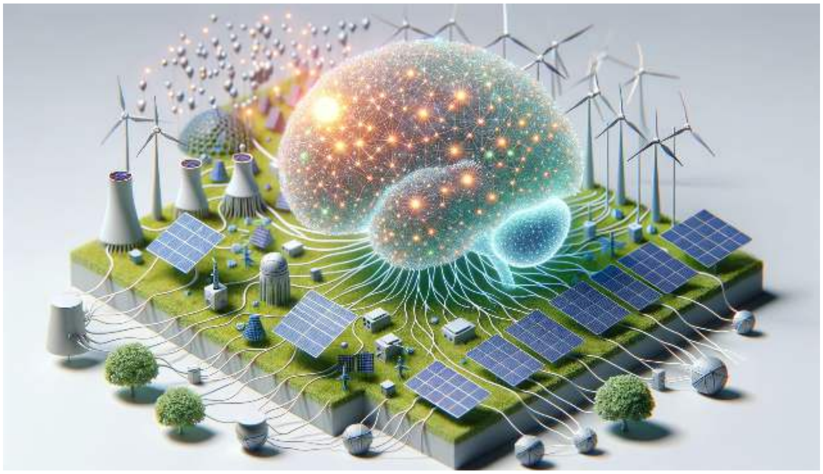


Figure 17.1: *DALL·E 3 Prompt: 3D illustration on a light background of a sustainable AI network interconnected with a myriad of eco-friendly energy sources. The AI actively manages and optimizes its energy from sources like solar arrays, wind turbines, and hydro dams, emphasizing power efficiency and performance. Deep neural networks spread throughout, receiving energy from these sustainable resources.*

Purpose

How do environmental considerations influence the design and implementation of machine learning systems, and what principles emerge from examining AI through an ecological perspective?

Machine learning systems inherently require significant computational resources, raising critical concerns about their environmental impact. Addressing these concerns requires a deep understanding of how architectural decisions affect energy consumption, resource utilization, and ecological sustainability. Designers and engineers must consider the relationships between computational demands, resource utilization, and environmental consequences across various system components. A systematic exploration of these considerations helps identify key architectural principles and design strategies that harmonize performance objectives with ecological stewardship.

Learning Objectives

- Define the key environmental impacts of AI systems.
- Identify the ethical considerations surrounding sustainable AI.
- Analyze strategies for reducing AI's carbon footprint.
- Describe the role of energy-efficient design in sustainable AI.
- Discuss the importance of policy and regulation for sustainable AI.
- Recognize key challenges in the AI hardware and software lifecycle.

17.1 Overview

Machine learning has become an essential driver of technological progress, powering advancements across industries and scientific domains. However, as AI models grow in complexity and scale, the computational demands required to train and deploy them have increased significantly, raising critical concerns about sustainability. The environmental impact of AI extends beyond energy consumption, encompassing carbon emissions, resource extraction, and electronic waste. As a result, it is imperative to examine AI systems through the lens of sustainability and assess the trade-offs between performance and ecological responsibility.

Developing large-scale AI models, such as state-of-the-art language and vision models, requires substantial computational power. Training a single large model can consume thousands of megawatt-hours of electricity, equivalent to powering hundreds of households for a month. Much of this energy is supplied by data centers, which rely heavily on nonrenewable energy sources, contributing to global carbon emissions. Estimates indicate that AI-related emissions are comparable to those of entire industrial sectors, highlighting the urgency of transitioning to more energy-efficient models and renewable-powered infrastructure.

Beyond energy consumption, AI systems also impact the environment through hardware manufacturing and resource utilization. Training and inference workloads depend on specialized processors, such as GPUs and TPUs, which require rare earth metals whose extraction and processing generate significant pollution. Additionally, the growing demand for AI applications accelerates electronic waste production, as hardware rapidly becomes obsolete. Even small-scale AI systems, such as those deployed on edge devices, contribute to sustainability challenges, necessitating careful consideration of their lifecycle impact.

This chapter examines the sustainability challenges associated with AI systems and explores emerging solutions to mitigate their environmental footprint. It discusses strategies for improving algorithmic efficiency, optimizing training infrastructure, and designing energy-efficient hardware. Additionally, it considers the role of renewable energy sources, regulatory frameworks, and industry best practices in promoting sustainable AI development. By addressing these challenges, the field can advance toward more ecologically responsible AI systems while maintaining technological progress.

17.2 Ethical Responsibility

17.2.1 Long-Term Viability

The long-term sustainability of AI is increasingly challenged by the exponential growth of computational demands required to train and deploy machine learning models. Over the past decade, AI systems have scaled at an unprecedented rate, with compute requirements increasing 350,000× from 2012 to 2019 (Schwartz et al. 2020). This trend shows no signs of slowing down, as advancements in deep learning continue to prioritize larger models with more parameters, larger training datasets, and higher computational complexity. However, sustaining this trajectory poses significant sustainability challenges, particularly as the efficiency gains from hardware improvements fail to keep pace with the rising demands of AI workloads.

Historically, computational efficiency improved with advances in semiconductor technology. Moore's Law, which predicted that the number of transistors on a chip would double approximately every two years, led to continuous improvements in processing power and energy efficiency. However, Moore's Law is now reaching fundamental physical limits, making further transistor scaling increasingly difficult and costly. Dennard scaling, which once ensured that smaller transistors would operate at lower power levels, has also ended, leading to stagnation in energy efficiency improvements per transistor. As a result, while AI models continue to scale in size and capability, the hardware running these models is no longer improving at the same exponential rate. This growing divergence between computational demand and hardware efficiency creates an unsustainable trajectory in which AI consumes ever-increasing amounts of energy.

The training of complex AI systems like large deep learning models demands startlingly high levels of computing power with profound energy implications. Consider OpenAI's state-of-the-art language model GPT-3 as a prime example. This system pushes the frontiers of text generation through algorithms trained on massive datasets, with training estimated to require 1,300 megawatt-hours (MWh) of electricity—roughly equivalent to the monthly energy consumption of 1,450 average U.S. households (Maslej et al. 2023). In recent years, these generative AI models have gained increasing popularity, leading to more models being trained with ever-growing parameter counts.

Research shows that increasing model size, dataset size, and compute used for training improves performance smoothly with no signs of saturation (Kaplan et al. 2020), as evidenced in Figure 17.2 where test loss decreases as each of these three factors increases. Beyond training, AI-powered applications such as large-scale recommender systems and generative models require continuous inference at scale, consuming significant energy even after training completes. As AI adoption grows across industries from finance to healthcare to entertainment, the cumulative energy burden of AI workloads continues to rise, raising concerns about the environmental impact of widespread deployment.

Beyond electricity consumption, the sustainability challenges of AI extend to hardware resource demands. High-performance computing (HPC) clusters and AI accelerators rely on specialized hardware, including GPUs, TPUs, and FPGAs, all of which require rare earth metals and complex manufacturing

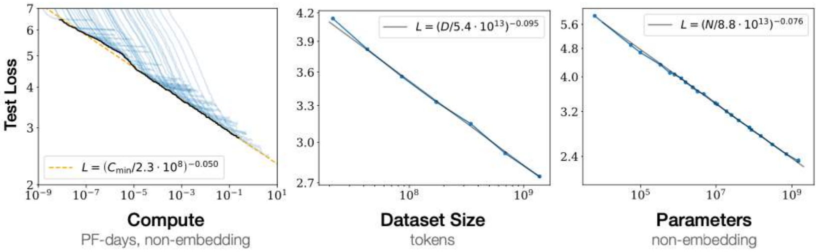


Figure 17.2: Performance improves with compute, dataset set, and model size. Source: Kaplan et al. (2020).

processes. The production of AI chips is energy-intensive, involving multiple fabrication steps that contribute to Scope 3 emissions, which account for the majority of the carbon footprint in semiconductor manufacturing. As model sizes continue to grow, the demand for AI hardware increases, further exacerbating the environmental impact of semiconductor production and disposal.

The long-term sustainability of AI requires a shift in how machine learning systems are designed, optimized, and deployed. As compute demands outpace efficiency improvements, addressing AI's environmental impact will require rethinking system architecture, energy-aware computing, and lifecycle management. Without intervention, the unchecked growth of AI models will continue to place unsustainable pressures on energy grids, data centers, and natural resources, underscoring the need for a more systematic approach to sustainable AI development.

The environmental impact of AI is not just a technical issue but also an ethical and social one. As AI becomes more integrated into our lives and industries, its sustainability becomes increasingly critical.

17.2.2 Ethical Issues

The environmental impact of AI raises fundamental ethical questions regarding the responsibility of developers, organizations, and policymakers to mitigate its carbon footprint. As AI systems continue to scale, their energy consumption and resource demands have far-reaching implications, necessitating a proactive approach to sustainability. Developers and companies that build and deploy AI systems must consider not only performance and efficiency but also the broader environmental consequences of their design choices.

A key ethical challenge lies in balancing technological progress with ecological responsibility. The pursuit of increasingly large models often prioritizes accuracy and capability over energy efficiency, leading to substantial environmental costs. While optimizing for sustainability may introduce trade-offs, including increased development time or minor reductions in accuracy, it is an ethical imperative to integrate environmental considerations into AI system design. This requires shifting industry norms toward sustainable computing practices, such as energy-aware training techniques, low-power hardware designs, and carbon-conscious deployment strategies (D. Patterson et al. 2021b).

Beyond sustainability, AI development also raises broader ethical concerns related to transparency, fairness, and accountability. Figure 17.3 illustrates the ethical challenges associated with AI development, linking different types of concerns, including inscrutable evidence, unfair outcomes, and traceability, to issues like opacity, bias, and automation bias. These concerns extend to sustainability, as the environmental trade-offs of AI development are often opaque and difficult to quantify. The lack of traceability in energy consumption and carbon emissions can lead to unjustified actions, where companies prioritize performance gains without fully understanding or disclosing the environmental costs.

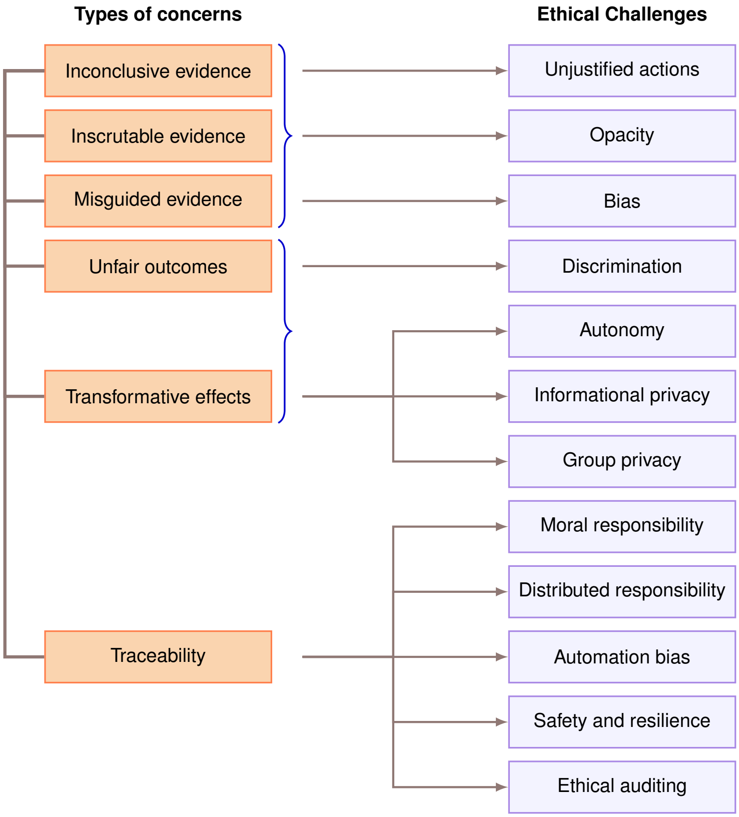


Figure 17.3: Ethical challenges in AI development. Source: COE

Addressing these concerns also demands greater transparency and accountability from AI companies. Large technology firms operate extensive cloud infrastructures that power modern AI applications, yet their environmental impact is often opaque. Organizations must take active steps to measure, report, and reduce their carbon footprint across the entire AI lifecycle, from hardware manufacturing to model training and inference. Voluntary self-regulation is an important first step, but policy interventions and industry-wide standards may be necessary to ensure long-term sustainability. Reported metrics such as energy consumption, carbon emissions, and efficiency benchmarks could serve as mechanisms to hold organizations accountable.

Furthermore, ethical AI development must encourage open discourse on environmental trade-offs. Researchers should be empowered to advocate for sustainability within their institutions and organizations, ensuring that environmental concerns are factored into AI development priorities. The broader AI community has already begun addressing these issues, as exemplified by the open letter advocating a pause on large-scale AI experiments, which highlights concerns about unchecked expansion. By fostering a culture of transparency and ethical responsibility, the AI industry can work toward aligning technological advancement with ecological sustainability.

AI has the potential to reshape industries and societies, but its long-term viability depends on how responsibly it is developed. Ethical AI development is not only about preventing harm to individuals and communities but also about ensuring that AI-driven innovation does not come at the cost of environmental degradation. As stewards of these powerful technologies, developers and organizations have a profound duty to integrate sustainability into AI's future trajectory.

17.2.3 Case Study: DeepMind's Energy Efficiency

Google's data centers form the backbone of services such as Search, Gmail, and YouTube, handling billions of queries daily. These data centers operate at massive scales, consuming vast amounts of electricity, particularly for cooling infrastructure that ensures optimal server performance. Improving the energy efficiency of data centers has long been a priority, but conventional engineering approaches faced diminishing returns due to the complexity of the cooling systems and the highly dynamic nature of environmental conditions. To address these challenges, Google collaborated with DeepMind to develop a machine learning-driven optimization system that could automate and enhance energy management at scale.

Building on more than a decade of efforts to optimize data center design, energy-efficient hardware, and renewable energy integration, DeepMind's AI approach targeted one of the most energy-intensive aspects of data centers: cooling systems. Traditional cooling relies on manually set heuristics that account for factors such as server heat output, external weather conditions, and architectural constraints. However, these systems exhibit nonlinear interactions, meaning that simple rule-based optimizations often fail to capture the full complexity of their operations. The result was suboptimal cooling efficiency, leading to unnecessary energy waste.

DeepMind's team trained a neural network model using Google's historical sensor data, which included real-time temperature readings, power consumption levels, cooling pump activity, and other operational parameters. The model learned the intricate relationships between these factors and could dynamically predict the most efficient cooling configurations. Unlike traditional approaches, which relied on human engineers periodically adjusting system settings, the AI model continuously adapted in real time to changing environmental and workload conditions.

The results were unprecedented efficiency gains. When deployed in live data center environments, DeepMind's AI-driven cooling system reduced cooling energy consumption by 40%, leading to an overall 15% improvement in Power Usage Effectiveness (PUE)—a key metric for data center energy efficiency that measures the ratio of total energy consumption to the energy used purely for computing tasks (Barroso, Hölzle, and Ranganathan 2019). Notably, these improvements were achieved without any additional hardware modifications, demonstrating the potential of software-driven optimizations to significantly reduce AI's carbon footprint.

Beyond a single data center, DeepMind's AI model provided a generalizable framework that could be adapted to different facility designs and climate conditions, offering a scalable solution for optimizing power consumption across global data center networks. This case study exemplifies how AI can be leveraged not just as a consumer of computational resources but as a tool for sustainability, driving substantial efficiency improvements in the infrastructure that supports machine learning itself.

The integration of data-driven decision-making, real-time adaptation, and scalable AI models demonstrates the growing role of intelligent resource management in sustainable AI system design. This breakthrough exemplifies how machine learning can be applied to optimize the very infrastructure that powers it, ensuring a more energy-efficient future for large-scale AI deployments.

17.3 AI Carbon Footprint

The carbon footprint of artificial intelligence is a critical aspect of its overall environmental impact. As AI adoption continues to expand, so does its energy consumption and associated greenhouse gas emissions. Training and deploying AI models require vast computational resources, often powered by energy-intensive data centers that contribute significantly to global carbon emissions. However, the carbon footprint of AI extends beyond electricity usage, encompassing hardware manufacturing, data storage, and end-user interactions—all of which contribute to emissions across an AI system's lifecycle.

Quantifying the carbon impact of AI is complex, as it depends on multiple factors, including the size of the model, the duration of training, the hardware used, and the energy sources powering data centers. Large-scale AI models, such as GPT-3, require thousands of megawatt-hours (MWh) of electricity, equivalent to the energy consumption of entire communities. The energy required for inference, the phase during which trained models produce outputs, is also substantial, particularly for widely deployed AI services such as real-time translation, image generation, and personalized recommendations. Unlike traditional software, which has a relatively static energy footprint, AI models consume energy continuously, leading to an ongoing sustainability challenge.

Beyond direct energy use, the carbon footprint of AI must also account for indirect emissions from hardware production and supply chains. Manufacturing AI accelerators such as GPUs, TPUs, and custom chips involves energy-intensive fabrication processes that rely on rare earth metals and complex supply chains. The full life cycle emissions of AI systems, which encompass data centers, hardware manufacturing, and global AI deployments, must be considered to develop more sustainable AI practices.

Understanding AI's carbon footprint requires breaking down where emissions come from, how they are measured, and what strategies can be employed to mitigate them. We explore the following:

- Carbon emissions and energy consumption trends in AI, which quantify AI's energy demand and provide real-world comparisons.
- Scopes of carbon emissions (Scope 1, 2, and 3)—differentiating between direct, indirect, and supply chain-related emissions.
- The energy cost of training vs. inference—analyzing how different phases of AI impact sustainability.

By dissecting these components, we can better assess the true environmental impact of AI systems and identify opportunities to reduce their footprint through more efficient design, energy-conscious deployment, and sustainable infrastructure choices.

17.3.1 Emissions & Consumption

Artificial intelligence systems require vast computational resources, making them one of the most energy-intensive workloads in modern computing. The energy consumed by AI systems extends beyond the training of large models to include ongoing inference workloads, data storage, and communication across distributed computing infrastructure. As AI adoption scales across industries, understanding its energy consumption patterns and carbon emissions is critical for designing more sustainable machine learning infrastructure.

Data centers play a central role in AI's energy demands, consuming vast amounts of electricity to power compute servers, storage, and cooling systems. Without access to renewable energy, these facilities rely heavily on nonrenewable sources such as coal and natural gas, contributing significantly to global carbon emissions. Current estimates suggest that data centers produce up to 2% of total global CO_2 emissions—a figure that is closing in on the airline industry's footprint (Yanan Liu et al. 2020). The energy burden of AI is expected to grow exponentially due to three key factors: increasing data center capacity, rising AI training workloads, and surging inference demands (D. Patterson, Gonzalez, Holzle, et al. 2022). Without intervention, these trends risk making AI's environmental footprint unsustainably large (Thompson, Spanuth, and Matthews 2023).

17.3.1.1 Energy Demands in Data Centers

AI workloads are among the most compute-intensive operations in modern data centers. Companies such as Meta operate hyperscale data centers spanning multiple football fields in size, housing hundreds of thousands of AI-optimized servers. The training of large language models (LLMs) such as GPT-4 required over 25,000 Nvidia A100 GPUs running continuously for 90 to 100 days (S. Choi and Yoon 2024), consuming thousands of megawatt-hours (MWh) of electricity. These facilities rely on high-performance AI accelerators like NVIDIA DGX H100 units, each of which can draw up to 10.2 kW at peak power (Choquette 2023).

AI's rapid adoption is driving a significant increase in data center energy consumption. As shown in Figure 17.4, the energy demand of AI workloads is projected to substantially increase total data center energy use, especially after 2024. While efficiency gains have historically offset rising power needs, these gains are decelerating, amplifying AI's environmental impact.

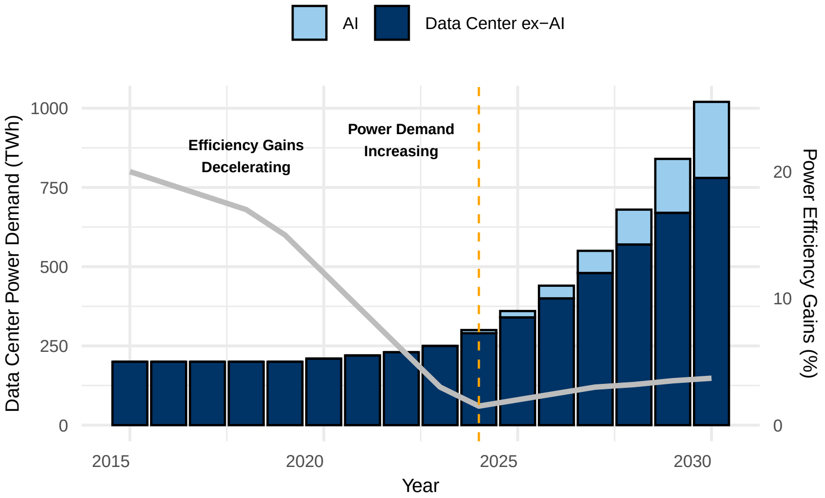


Figure 17.4: Projected Data Center and AI Power Demand with Power Efficiency Gains (2015–2030). Source: Masanet et al. (2020), Cisco, IEA, Goldman Sachs Global Investment Research.

Cooling is another major factor in AI's energy footprint. Large-scale AI training and inference workloads generate massive amounts of heat, necessitating advanced cooling solutions to prevent hardware failures. Estimates indicate that 30-40% of a data center's total electricity usage goes into cooling alone (Dayarathna, Wen, and Fan 2016). Companies have begun adopting alternative cooling methods to reduce this demand. For example, Microsoft's data center in Ireland leverages a nearby fjord, using over half a million gallons of seawater daily to dissipate heat. However, as AI models scale in complexity, cooling demands continue to grow, making sustainable AI infrastructure design a pressing challenge.

17.3.1.2 AI vs. Other Industries

The environmental impact of AI workloads has emerged as a significant concern, with carbon emissions approaching levels comparable to established carbon-intensive sectors. Research demonstrates that training a single large AI model generates carbon emissions equivalent to multiple passenger vehicles over their complete lifecycle (Strubell, Ganesh, and McCallum 2019b). To contextualize AI's environmental footprint, Figure 17.6 compares the carbon emissions of large-scale machine learning tasks to transcontinental flights, illustrating the substantial energy demands of training and inference workloads. It shows a comparison from lowest to highest carbon footprints, starting with a roundtrip flight between NY and SF, human life average per year, American life average per year, US car including fuel over a lifetime, and a Transformer model with neural architecture search, which has the highest footprint. These comparisons underscore the need for more sustainable AI practices to mitigate the industry's carbon impact.

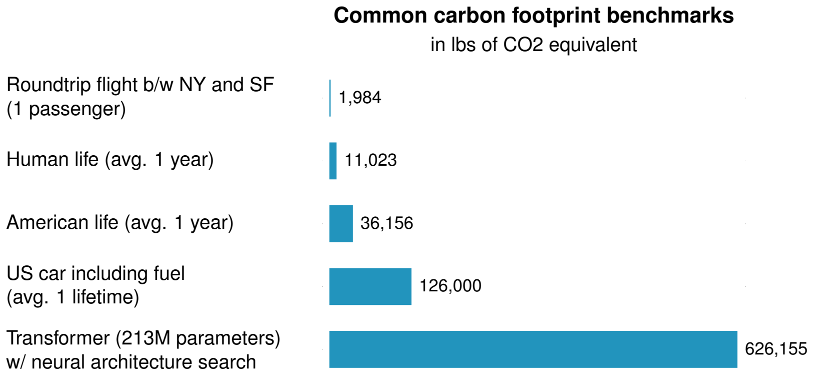


Figure 17.5: Carbon footprint of NLP model in lbs of carbon dioxide.

The training phase of large natural language processing models produces carbon dioxide emissions comparable to hundreds of transcontinental flights. When examining the broader industry impact, AI's aggregate computational carbon footprint is approaching parity with the commercial aviation sector. Furthermore, as AI applications scale to serve billions of users globally, the cumulative emissions from continuous inference operations may ultimately exceed those generated during training.

Figure 17.6 provides a detailed analysis of carbon emissions across various large-scale machine learning tasks at Meta, illustrating the substantial environmental impact of different AI applications and architectures. This quantitative assessment of AI's carbon footprint underscores the pressing need to develop more sustainable approaches to machine learning development and deployment. Understanding these environmental costs is crucial for implementing effective mitigation strategies and advancing the field responsibly.

17.3.2 Updated Analysis

Moreover, AI's impact extends beyond energy consumption during operation. The full lifecycle emissions of AI include hardware manufacturing, supply chain emissions, and end-of-life disposal, making AI a significant contributor to environmental degradation. AI models not only require electricity to train and infer, but they also depend on a complex infrastructure of semiconductor fabrication, rare earth metal mining[0], and electronic waste disposal. The next section breaks down AI's carbon emissions into Scope 1 (direct emissions), Scope 2 (indirect emissions from electricity), and Scope 3 (supply chain and lifecycle emissions) to provide a more detailed view of its environmental impact.

17.3.3 Carbon Emission Scopes

AI is expected to see an annual growth rate of 37.3% between 2023 and 2030. Yet, applying the same growth rate to operational computing could multiply

[0] The production of AI chips requires rare earth elements such as neodymium and dysprosium, the extraction of which has significant environmental consequences.

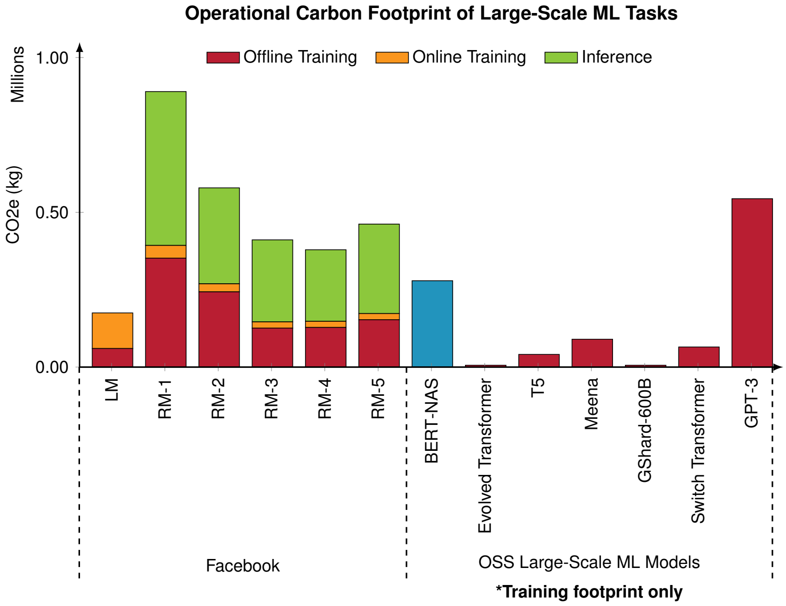


Figure 17.6: Carbon footprint of large-scale ML tasks. Source: C.-J. Wu et al. (2022).

annual AI energy needs up to 1,000 times by 2030. So, while model optimization tackles one facet, responsible innovation must also consider total lifecycle costs at global deployment scales that were unfathomable just years ago but now pose infrastructure and sustainability challenges ahead.

17.3.3.1 Scope 1

Scope 1 emissions refer to direct greenhouse gas emissions produced by AI data centers and computing facilities. These emissions result primarily from on-site power generation, including backup diesel generators used to ensure reliability in large cloud environments, as well as facility cooling systems. Although many AI data centers predominantly rely on grid electricity, those with their own power plants or fossil-fuel-dependent backup systems contribute significantly to direct emissions, especially in regions where renewable energy sources are less prevalent (Masanet et al. 2020a).

17.3.3.2 Scope 2

Scope 2 emissions encompass indirect emissions from electricity purchased to power AI infrastructure. The majority of AI's operational energy consumption falls under Scope 2, as cloud providers and enterprise computing facilities require massive electrical inputs for GPUs, TPUs, and high-density servers. The carbon intensity associated with Scope 2 emissions varies geographically based on regional energy mixes. Regions dominated by coal and natural gas electricity generation create significantly higher AI-related emissions compared to regions utilizing renewable sources such as wind, hydro, or solar. This geographic variability motivates companies to strategically position data centers in areas with cleaner energy sources and adopt carbon-aware scheduling strategies to reduce emissions (Alvim et al. 2022).

17.3.3.3 Scope 3

Scope 3 emissions constitute the largest and most complex category, capturing indirect emissions across the entire AI supply chain and lifecycle. These emissions originate from manufacturing, transportation, and disposal of AI hardware, particularly semiconductors and memory modules. Semiconductor manufacturing is particularly energy-intensive, involving complex processes such as chemical etching, rare-earth metal extraction, and extreme ultraviolet (EUV) lithography, all of which produce substantial carbon outputs. Indeed, manufacturing a single high-performance AI accelerator can generate emissions equivalent to several years of operational energy use (U. Gupta, Kim, et al. 2022).

Beyond manufacturing, Scope 3 emissions include the downstream impact of AI once deployed. AI services such as search engines, social media platforms, and cloud-based recommendation systems operate at enormous scale, requiring continuous inference across millions or even billions of user interactions. The cumulative electricity demand of inference workloads can ultimately surpass the energy used for training, further amplifying AI's carbon impact. End-user devices, including smartphones, IoT devices, and edge computing platforms, also contribute to Scope 3 emissions, as their AI-enabled functionality depends on sustained computation. Companies such as Meta and Google report that Scope 3 emissions from AI-powered services make up the largest share of their total environmental footprint, due to the sheer scale at which AI operates.

These massive facilities provide the infrastructure for training complex neural networks on vast datasets. For instance, based on leaked information, OpenAI's language model GPT-4 was trained on Azure data centers packing over 25,000 Nvidia A100 GPUs, used continuously for over 90 to 100 days.

The GHG Protocol framework, illustrated in Figure 17.7, provides a structured way to visualize the sources of AI-related carbon emissions. Scope 1 emissions arise from direct company operations, such as data center power generation and company-owned infrastructure. Scope 2 covers electricity purchased from the grid, the primary source of emissions for cloud computing workloads. Scope 3 extends beyond an organization's direct control, including emissions from hardware manufacturing, transportation, and even the end-user energy consumption of AI-powered services. Understanding this breakdown allows for more targeted sustainability strategies, ensuring that efforts to reduce AI's environmental impact are not solely focused on energy efficiency but also address the broader supply chain and lifecycle emissions that contribute significantly to the industry's carbon footprint.

17.3.4 Training vs. Inference Impact

The energy consumption of AI systems is often closely associated with the training phase, where substantial computational resources are utilized to develop large-scale machine learning models. However, while training demands significant power, it represents a one-time cost per model version. In contrast,

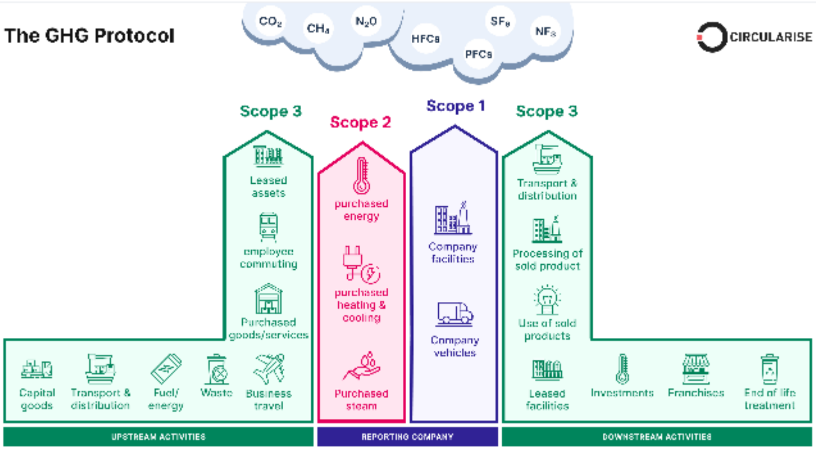


Figure 17.7: The GHG Protocol framework categorizes emissions into Scope 1, 2, and 3, helping organizations assess their direct and indirect carbon impact. Source: Circularise.

inference, which involves the continuous application of trained models to new data, happens continuously at a massive scale and often becomes the dominant contributor to energy consumption over time (D. Patterson et al. 2021b). As AI-powered services, such as real-time translation, recommender systems, and generative AI applications expand globally, inference workloads increasingly drive AI's overall carbon footprint.

17.3.4.1 Training Energy Demands

Training state-of-the-art AI models demands enormous computational resources. For example, models like GPT-4 were trained using over 25,000 Nvidia A100 GPUs operating continuously for approximately three months within cloud-based data centers (S. Choi and Yoon 2024). OpenAI's dedicated supercomputer infrastructure, built specifically for large-scale AI training, contains 285,000 CPU cores, 10,000 GPUs, and network bandwidth exceeding 400 gigabits per second per server, illustrating the vast scale and associated energy consumption of AI training infrastructures (D. Patterson et al. 2021b).

High-performance AI accelerators, such as NVIDIA DGX H100 systems, are specifically designed for these training workloads. Each DGX H100 unit can draw up to 10.2 kW at peak load, with clusters often consisting of thousands of nodes running continuously (Choquette 2023). The intensive computational loads result in significant heat dissipation, necessitating substantial cooling infrastructure. Cooling alone can account for 30-40% of total data center energy consumption (Dayarathna, Wen, and Fan 2016).

While significant, these energy costs occur once per trained model. The primary sustainability challenge emerges during model deployment, where inference workloads continuously serve millions or billions of users.

17.3.4.2 Inference Energy Costs

Inference workloads execute every time an AI model responds to queries, classifies images, or makes predictions. Unlike training, inference scales dynamically and continuously across applications such as search engines, recommendation systems, and generative AI models. Although each individual inference request consumes far less energy compared to training, the cumulative energy usage from billions of daily AI interactions quickly surpasses training-related consumption (D. Patterson et al. 2021b).

For example, AI-driven search engines handle billions of queries per day, recommendation systems provide personalized content continuously, and generative AI services such as ChatGPT or DALL-E have substantial per-query computational costs. The inference energy footprint is especially pronounced in transformer-based models due to high memory and computational bandwidth requirements.

As shown in Figure 17.8, the market for inference workloads in data centers is projected to grow significantly from $4-5 billion in 2017 to $9-10 billion by 2025, more than doubling in size. Similarly, edge inference workloads are expected to increase from less than $0.1 billion to $4-4.5 billion in the same period. This growth substantially outpaces the expansion of training workloads in both environments, highlighting how the economic footprint of inference is rapidly outgrowing that of training operations.

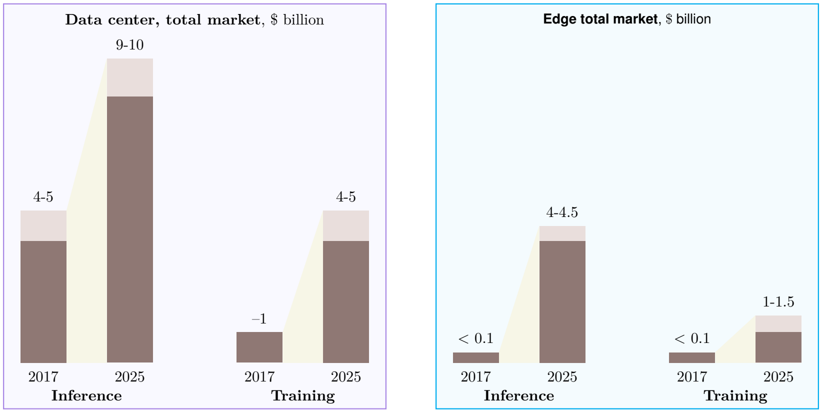


Figure 17.8: Market size for inference and training hardware. Source: McKinsey.

Unlike traditional software applications with fixed energy footprints, inference workloads dynamically scale with user demand. AI services like Alexa, Siri, and Google Assistant rely on continuous cloud-based inference, processing millions of voice queries per minute, necessitating uninterrupted operation of energy-intensive data center infrastructure.

17.3.4.3 Edge AI Impact

Inference does not always happen in large data centers—edge AI is emerging as a viable alternative to reduce cloud dependency. Instead of routing every AI request to centralized cloud servers, some AI models can be deployed directly on user devices or at edge computing nodes. This approach reduces data transmission energy costs and lowers the dependency on high-power cloud inference.

However, running inference at the edge does not eliminate energy concerns—especially when AI is deployed at scale. Autonomous vehicles, for instance, require millisecond-latency AI inference, meaning cloud processing is impractical. Instead, vehicles are now being equipped with onboard AI accelerators that function as "data centers on wheels (Sudhakar, Sze, and Karaman 2023). These embedded computing systems process real-time sensor data equivalent to small data centers, consuming significant power even without relying on cloud inference.

Similarly, consumer devices such as smartphones, wearables, and IoT sensors individually consume relatively little power but collectively contribute significantly to global energy use due to their sheer numbers. Therefore, the efficiency benefits of edge computing must be balanced against the extensive scale of device deployment.

17.4 Beyond Carbon

While reducing AI's carbon emissions is critical, the environmental impact extends far beyond energy consumption. The manufacturing of AI hardware involves significant resource extraction, hazardous chemical usage, and water consumption that often receive less attention despite their ecological significance.

Modern semiconductor fabrication plants (fabs) that produce AI chips require millions of gallons of water daily and use over 250 hazardous substances in their processes (Mills and Le Hunte 1997). In regions already facing water stress, such as Taiwan, Arizona, and Singapore, this intensive water usage threatens local ecosystems and communities.

The industry also relies heavily on scarce materials like gallium, indium, arsenic, and helium, which are essential for AI accelerators and high-speed communication chips (H.-W. Chen 2006; M. Davies 2011). These materials face both geopolitical supply risks and depletion concerns.

We will explore these critical but often overlooked aspects of AI's environmental impact, including water consumption, hazardous waste production, rare material scarcity, and biodiversity disruption. Understanding these broader ecological impacts is essential for developing truly sustainable AI infrastructure.

17.4.1 Water Usage

Semiconductor fabrication is an exceptionally water-intensive process, requiring vast quantities of ultrapure water [1] for cleaning, cooling, and chemical processing. The scale of water consumption in modern fabs is comparable to that of entire urban populations. For example, TSMC's latest fab in Arizona is projected to consume 8.9 million gallons of water per day, accounting for nearly 3% of the city's total water production. This demand places significant strain on local water resources, particularly in water-scarce regions such as Taiwan, Arizona, and Singapore, where semiconductor manufacturing is concentrated. Semiconductor companies have recognized this challenge and are actively investing in recycling technologies and more efficient water management practices. STMicroelectronics, for example, recycles and reuses approximately 41%

[1] Ultrapure water (UPW): Water that has been purified to stringent standards, typically containing less than 1 part per billion of impurities. UPW is essential for semiconductor fabrication, as even trace contaminants can impair chip performance and yield.

of its water, significantly reducing its environmental footprint (see Figure 17.9 showing the typical semiconductor fab water cycle).

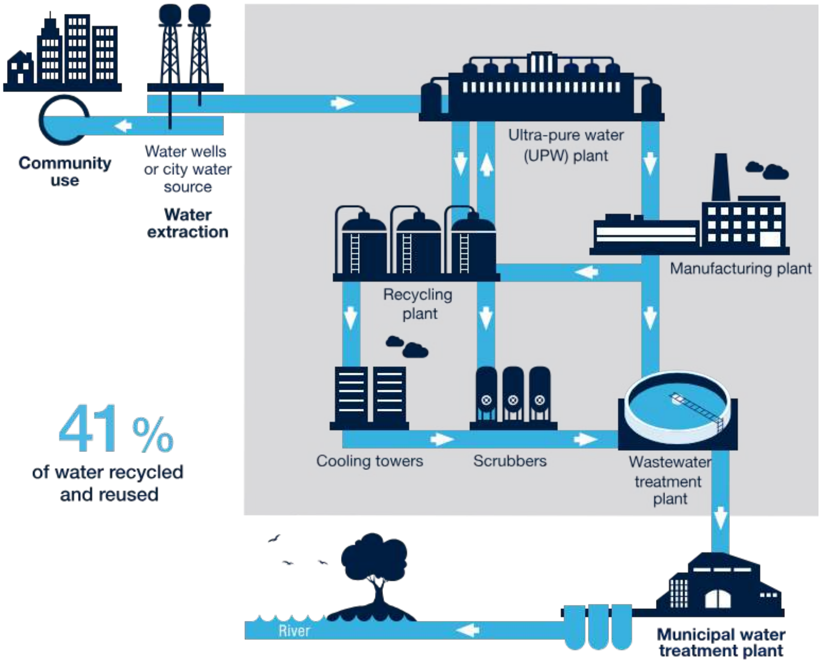

Community use
Water wells or city water source
Water extraction
Ultra-pure water (UPW) plant
Manufacturing plant
Recycling plant
41 %
of water recycled and reused
Cooling towers
Scrubbers
Wastewater treatment plant
River
Municipal water treatment plant

Figure 17.9: Typical water cycle in semiconductor manufacturing, illustrating water extraction, ultrapure water generation, manufacturing use, recycling processes, and wastewater treatment before discharge or reuse. Source: ST Sustainability Report.

The primary use of ultrapure water in semiconductor fabrication is for flushing contaminants from wafers at various production stages. Water also serves as a coolant and carrier fluid in thermal oxidation, chemical deposition, and planarization processes. A single 300mm silicon wafer requires over 8,300 liters of water, with more than two-thirds of this being ultrapure water (Cope 2009). During peak summer months, the cumulative daily water consumption of major fabs rivals that of cities with populations exceeding half a million people.

The impact of this massive water usage extends beyond consumption. Excessive water withdrawal from local aquifers lowers groundwater levels, leading to issues such as land subsidence and saltwater intrusion[2]. In Hsinchu, Taiwan, one of the world's largest semiconductor hubs, extensive water extraction by fabs has led to falling water tables and encroaching seawater contamination, affecting both agriculture and drinking water supplies.

Figure 17.10 contextualizes the daily water footprint of data centers compared to other industrial uses, illustrating the immense water demand of high-tech infrastructure.

While some semiconductor manufacturers implement water recycling systems, the effectiveness of these measures varies. Intel reports that 97% of its direct water consumption is attributed to fabrication processes (Cooper et al. 2011), and while water reuse is increasing, the sheer scale of water withdrawals remains a critical sustainability challenge.

[2] Saltwater Intrusion: The process by which seawater enters freshwater aquifers due to groundwater overuse, leading to water quality degradation.

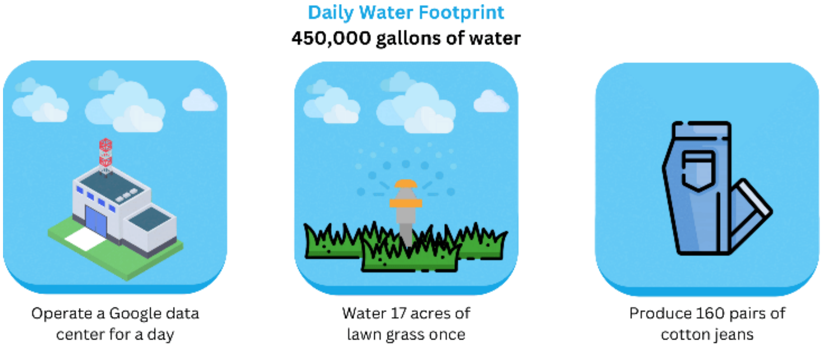

Daily Water Footprint
450,000 gallons of water
Operate a Google data center for a day
Water 17 acres of lawn grass once
Produce 160 pairs of cotton jeans

Figure 17.10: Daily Water Footprint of Datacenters in comparison with other water uses. Source: Google's Data Center Cooling

Beyond depletion, water discharge from semiconductor fabs introduces contamination risks if not properly managed. Wastewater from fabrication contains metals, acids, and chemical residues that must be thoroughly treated before release. Although modern fabs employ advanced purification systems, the extraction of contaminants still generates hazardous byproducts, which, if not carefully disposed of, pose risks to local ecosystems.

The growing demand for semiconductor manufacturing, driven by AI acceleration and computing infrastructure expansion, makes water management a crucial factor in sustainable AI development. Ensuring the long-term viability of semiconductor production requires not only reducing direct water consumption but also enhancing wastewater treatment and developing alternative cooling technologies that minimize reliance on fresh water sources.

17.4.2 Hazardous Chemicals

Semiconductor fabrication is heavily reliant on highly hazardous chemicals, which play an essential role in processes such as etching, doping, and wafer cleaning. The manufacturing of AI hardware, including GPUs, TPUs, and other specialized accelerators, requires the use of strong acids, volatile solvents, and toxic gases, all of which pose significant health and environmental risks if not properly managed. The scale of chemical usage in fabs is immense, with thousands of metric tons of hazardous substances consumed annually (S. Kim et al. 2018).

Among the most critical chemical categories used in fabrication are strong acids, which facilitate wafer etching and oxide removal. Hydrofluoric acid, sulfuric acid, nitric acid, and hydrochloric acid are commonly employed in the cleaning and patterning stages of chip production. While effective for these processes, these acids are highly corrosive and toxic, capable of causing severe chemical burns and respiratory damage if mishandled. Large semiconductor fabs require specialized containment, filtration, and neutralization systems to prevent accidental exposure and environmental contamination.

Solvents are another critical component in chip manufacturing, primarily used for dissolving photoresists and cleaning wafers. Key solvents include xylene, methanol, and methyl isobutyl ketone (MIBK), which, despite their utility, present air pollution and worker safety risks. These solvents are volatile organic compounds (VOCs)[3] that can evaporate into the atmosphere, contributing to indoor and outdoor air pollution. If not properly contained, VOC exposure can result in neurological damage, respiratory issues, and long-term health effects for workers in semiconductor fabs.

Toxic gases are among the most dangerous substances used in AI chip manufacturing. Gases such as arsine (AsH_3), phosphine (PH_3), diborane (B_2H_6), and germane (GeH_4) are used in doping and chemical vapor deposition processes, essential for fine-tuning semiconductor properties. These gases are highly toxic and even fatal at low concentrations, requiring extensive handling precautions, gas scrubbers, and emergency response protocols. Any leaks or accidental releases in fabs can lead to severe health hazards for workers and surrounding communities.

While modern fabs employ strict safety controls, protective equipment, and chemical treatment systems, incidents still occur, leading to chemical spills, gas leaks, and contamination risks. The challenge of effectively managing hazardous chemicals is heightened by the ever-increasing complexity of AI accelerators, which require more advanced fabrication techniques and new chemical formulations.

Beyond direct safety concerns, the long-term environmental impact of hazardous chemical use remains a major sustainability issue. Semiconductor fabs generate large volumes of chemical waste, which, if improperly handled, can contaminate groundwater, soil, and local ecosystems. Regulations in many countries require fabs to neutralize and treat waste before disposal, but compliance and enforcement vary globally, leading to differing levels of environmental protection.

To mitigate these risks, fabs must continue advancing green chemistry initiatives, exploring alternative etchants, solvents, and gas formulations that reduce toxicity while maintaining fabrication efficiency. Additionally, process optimizations that minimize chemical waste, improve containment, and enhance recycling efforts will be essential to reducing the environmental footprint of AI hardware production.

17.4.3 Resource Depletion

While silicon is abundant and readily available, the fabrication of AI accelerators, GPUs, and specialized AI chips depends on scarce and geopolitically sensitive materials that are far more difficult to source. AI hardware manufacturing requires a range of rare metals, noble gases, and semiconductor compounds, many of which face supply constraints, geopolitical risks, and environmental extraction costs. As AI models become larger and more computationally intensive, the demand for these materials continues to rise, raising concerns about long-term availability and sustainability.

Although silicon forms the foundation of semiconductor devices, high-performance AI chips depend on rare elements such as gallium, indium, and arsenic, which are essential for high-speed, low-power electronic components (H.-W. Chen 2006). Gallium and indium, for example, are widely used in compound semiconductors, particularly for 5G communications, optoelectronics, and AI accelerators. The United States Geological Survey (USGS) has classified indium as a

[3] Volatile organic compounds (VOCs): Organic chemicals that easily evaporate into the air, posing health risks and contributing to air pollution. VOCs are commonly used in semiconductor manufacturing for cleaning, etching, and photoresist removal.

critical material, with global supplies expected to last fewer than 15 years at the current rate of consumption (M. Davies 2011).

Another major concern is helium, a noble gas critical for semiconductor cooling, plasma etching, and EUV lithography[4] used in next-generation chip production. Helium is unique in that once released into the atmosphere, it escapes Earth's gravity and is lost forever, making it a non-renewable resource (M. Davies 2011). The semiconductor industry is one of the largest consumers of helium, and supply shortages have already led to price spikes and disruptions in fabrication processes. As AI hardware manufacturing scales, the demand for helium will continue to grow, necessitating more sustainable extraction and recycling practices.

Beyond raw material availability, the geopolitical control of rare earth elements poses additional challenges. China currently dominates over 90% of the world's rare earth element (REE) refining capacity, including materials essential for AI chips, such as neodymium (for high-performance magnets in AI accelerators) and yttrium (for high-temperature superconductors) (A. R. Jha 2014). This concentration of supply creates supply chain vulnerabilities, as trade restrictions or geopolitical tensions could severely impact AI hardware production.

Table 17.1 highlights the key materials essential for AI semiconductor manufacturing, their applications, and supply concerns.

Table 17.1: Rare materials that are widely used in the semiconductor industry that are facing resource depletion.

Material	Application in AI Semiconductor Manufacturing	Supply Concerns
Silicon (Si)	Primary substrate for chips, wafers, transistors	Processing constraints; geopolitical risks
Gallium (Ga)	GaN-based power amplifiers, high-frequency components	Limited availability; byproduct of aluminum and zinc production
Germanium (Ge)	High-speed transistors, photodetectors, optical interconnects	Scarcity; geographically concentrated
Indium (In)	Indium Tin Oxide (ITO), optoelectronics	Limited reserves; recycling dependency
Tantalum (Ta)	Capacitors, stable integrated components	Conflict mineral; vulnerable supply chains
Rare Earth Elements (REEs)	Magnets, sensors, high-performance electronics	High geopolitical risks; environmental extraction concerns
Cobalt (Co)	Batteries for edge computing devices	Human rights issues; geographical concentration (Congo)
Tungsten (W)	Interconnects, barriers, heat sinks	Limited production sites; geopolitical concerns
Copper (Cu)	Interconnects, barriers, heat sinks	Limited high-purity sources; geopolitical concerns
Helium (He)	Semiconductor cooling, plasma etching, EUV lithography	Non-renewable; irretrievable atmospheric loss; limited extraction capacity
Indium (In)	ITO layers, optoelectronic components	Limited global reserves; geopolitical concentration
Cobalt (Co)	Batteries for edge computing devices	Geographical concentration; human rights concerns
Tungsten (W)	Interconnects, heat sinks	Limited production sites; geopolitical concerns
Copper (Cu)	Conductive pathways, wiring	Geopolitical dependencies; limited recycling capacity

The rapid growth of AI and semiconductor demand has accelerated the depletion of these critical resources, creating an urgent need for material recycling, substitution strategies, and more sustainable extraction methods. Some efforts

[4] Extreme ultraviolet (EUV) lithography: A cutting-edge semiconductor manufacturing technique that uses EUV light to etch nanoscale features on silicon wafers. EUV lithography is essential for producing advanced AI chips with smaller transistors and higher performance.

are underway to explore alternative semiconductor materials that reduce dependency on rare elements, but these solutions require significant advancement before they become viable alternatives at scale.

17.4.4 Waste Generation

Semiconductor fabrication produces significant volumes of hazardous waste, including gaseous emissions, VOCs, chemical-laden wastewater, and solid toxic byproducts. The production of AI accelerators, GPUs, and other high-performance chips involves multiple stages of chemical processing, etching, and cleaning, each generating waste materials that must be carefully treated to prevent environmental contamination.

Fabs release gaseous waste from various processing steps, particularly chemical vapor deposition (CVD), plasma etching, and ion implantation. This includes toxic and corrosive gases such as arsine (AsH_3), phosphine (PH_3), and germane (GeH_4), which require advanced scrubber systems to neutralize before release into the atmosphere. If not properly filtered, these gases pose severe health hazards and contribute to air pollution and acid rain formation (Grossman 2007).

VOCs are another major waste category, emitted from photoresist processing, cleaning solvents, and lithographic coatings. Chemicals such as xylene, acetone, and methanol readily evaporate into the air, where they contribute to ground-level ozone formation and indoor air quality hazards for fab workers. In regions where semiconductor production is concentrated, such as Taiwan and South Korea, regulators have imposed strict VOC emission controls to mitigate their environmental impact.

Semiconductor fabs also generate large volumes of spent acids and metal-laden wastewater, requiring extensive treatment before discharge. Strong acids such as sulfuric acid, hydrofluoric acid, and nitric acid are used to etch silicon wafers, removing excess materials during fabrication. When these acids become contaminated with heavy metals, fluorides, and chemical residues, they must undergo neutralization and filtration before disposal. Improper handling of wastewater has led to groundwater contamination incidents, highlighting the importance of robust waste management systems (Prakash et al. 2023).

The solid waste produced in AI hardware manufacturing includes sludge, filter cakes, and chemical residues collected from fab exhaust and wastewater treatment systems. These byproducts often contain concentrated heavy metals, rare earth elements, and semiconductor process chemicals, making them hazardous for conventional landfill disposal. In some cases, fabs incinerate toxic waste, generating additional environmental concerns related to airborne pollutants and toxic ash disposal.

Beyond the waste generated during manufacturing, the end-of-life disposal of AI hardware presents another sustainability challenge. AI accelerators, GPUs, and server hardware have short refresh cycles, with data center equipment typically replaced every 3-5 years. This results in millions of tons of e-waste annually, much of which contains toxic heavy metals such as lead, cadmium, and mercury. Despite growing efforts to improve electronics recycling, current systems capture only 17.4% of global e-waste, leaving the majority to be discarded in landfills or improperly processed (Singh and Ogunseitan 2022).

Addressing the hazardous waste impact of AI requires advancements in both semiconductor manufacturing and e-waste recycling. Companies are exploring closed-loop recycling for rare metals, improved chemical treatment processes, and alternative materials with lower toxicity. However, as AI models continue to drive demand for higher-performance chips and larger-scale computing infrastructure, the industry's ability to manage its waste footprint will be a key factor in achieving sustainable AI development.

17.4.5 Biodiversity Impact

The environmental footprint of AI hardware extends beyond carbon emissions, resource depletion, and hazardous waste. The construction and operation of semiconductor fabrication facilities (fabs), data centers, and supporting infrastructure directly impact natural ecosystems, contributing to habitat destruction, water stress, and pollution. These environmental changes have far-reaching consequences for wildlife, plant ecosystems, and aquatic biodiversity, highlighting the need for sustainable AI development that considers broader ecological effects.

Semiconductor fabs and data centers require large tracts of land, often leading to deforestation and destruction of natural habitats. These facilities are typically built in industrial parks or near urban centers, but as demand for AI hardware increases, fabs are expanding into previously undeveloped regions, encroaching on forests, wetlands, and agricultural land.

The physical expansion of AI infrastructure disrupts wildlife migration patterns, as roads, pipelines, transmission towers, and supply chains fragment natural landscapes. Species that rely on large, connected ecosystems for survival, including migratory birds, large mammals, and pollinators, face increased barriers to movement, reducing genetic diversity and population stability. In regions with dense semiconductor manufacturing, such as Taiwan and South Korea, habitat loss has already been linked to declining biodiversity in affected areas (Hsu et al. 2016).

The massive water consumption of semiconductor fabs poses serious risks to aquatic ecosystems, particularly in water-stressed regions. Excessive groundwater extraction for AI chip production can lower water tables, affecting local rivers, lakes, and wetlands. In Hsinchu, Taiwan, where fabs draw millions of gallons of water daily, seawater intrusion has been reported in local aquifers, altering water chemistry and making it unsuitable for native fish species and vegetation.

Beyond depletion, wastewater discharge from fabs introduces chemical contaminants into natural water systems. While many facilities implement advanced filtration and recycling, even trace amounts of heavy metals, fluorides, and solvents can accumulate in water bodies, bioaccumulating in fish and disrupting aquatic ecosystems. Additionally, thermal pollution from data centers, which release heated water back into lakes and rivers, can raise temperatures beyond tolerable levels for native species, affecting oxygen levels and reproductive cycles (LeRoy Poff, Brinson, and Day 2002).

Semiconductor fabs emit a variety of airborne pollutants, including VOCs, acid mists, and metal particulates, which can travel significant distances before settling in the environment. These emissions contribute to air pollution and acid deposition, which damage plant life, soil quality, and nearby agricultural systems.

Airborne chemical deposition has been linked to tree decline, reduced crop yields, and soil acidification, particularly near industrial semiconductor hubs. In areas with high VOC emissions, plant growth can be stunted by prolonged exposure, affecting ecosystem resilience and food chains. Additionally, accidental chemical spills or gas leaks from fabs pose severe risks to both local wildlife and human populations, requiring strict regulatory enforcement to minimize long-term ecological damage (Wald and Jones 1987).

The environmental consequences of AI hardware manufacturing demonstrate the urgent need for sustainable semiconductor production, including reduced land use, improved water recycling, and stricter emissions controls. Without intervention, the accelerating demand for AI chips could further strain global biodiversity, emphasizing the importance of balancing technological progress with ecological responsibility.

17.5 Semiconductor Life Cycle

The environmental footprint of AI systems extends beyond energy consumption during model training and inference. A comprehensive assessment of AI's sustainability must consider the entire lifecycle—from the extraction of raw materials used in hardware manufacturing to the eventual disposal of obsolete computing infrastructure. Life Cycle Analysis (LCA) provides a systematic approach to quantifying the cumulative environmental impact of AI across its four key phases: design, manufacture, use, and disposal.

By applying LCA to AI systems, researchers and policymakers can pinpoint critical intervention points to reduce emissions, improve resource efficiency, and implement sustainable practices. This approach provides a holistic understanding of AI's ecological costs, extending sustainability considerations beyond operational power consumption to include hardware supply chains and electronic waste management.

Figure 17.11 illustrates the four primary stages of an AI system's lifecycle, each contributing to its total environmental footprint.

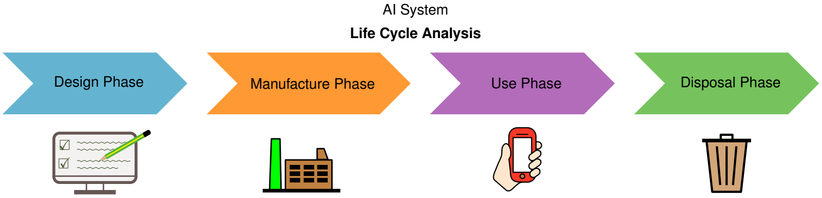


Figure 17.11: AI System Life Cycle Analysis is divided into four key phases: Design, Manufacture, Use, Disposal.

The following sections will analyze each lifecycle phase in detail, exploring its specific environmental impacts and sustainability challenges.

17.5.1 Design Phase

The design phase of an AI system encompasses the research, development, and optimization of machine learning models before deployment. This stage involves iterating on model architectures, adjusting hyperparameters, and running training experiments to improve performance. These processes are computationally intensive, requiring extensive use of hardware resources and energy. The environmental cost of AI model design is often underestimated, but repeated training runs, algorithm refinements, and exploratory experimentation contribute significantly to the overall sustainability impact of AI systems.

Developing an AI model requires running multiple experiments to determine the most effective architecture. Neural architecture search (NAS), for instance, automates the process of selecting the best model structure by evaluating hundreds or even thousands of configurations, each requiring a separate training cycle. Similarly, hyperparameter tuning involves modifying parameters such as learning rates, batch sizes, and optimization strategies to enhance model performance, often through exhaustive search techniques. Pre-training and fine-tuning further add to the computational demands, as models undergo multiple training iterations on different datasets before deployment. The iterative nature of this process results in high energy consumption, with studies indicating that hyperparameter tuning alone can account for up to 80% of training-related emissions (Strubell, Ganesh, and McCallum 2019b).

The scale of energy consumption in the design phase becomes evident when examining large AI models. OpenAI's GPT-3, for example, required an estimated 1,300 megawatt-hours (MWh) of electricity for training, a figure comparable to the energy consumption of 1,450 U.S. homes over an entire month (Maslej et al. 2023). However, this estimate only reflects the final training run and does not account for the extensive trial-and-error processes that preceded model selection. In deep reinforcement learning applications, such as DeepMind's AlphaZero, models undergo repeated training cycles to improve decision-making policies, further amplifying energy demands.

The carbon footprint of AI model design varies significantly depending on the computational resources required and the energy sources powering the data centers where training occurs. A widely cited study found that training a single large-scale natural language processing (NLP) model could produce emissions equivalent to the lifetime carbon footprint of five cars (Strubell, Ganesh, and McCallum 2019b). The impact is even more pronounced when training is conducted in data centers reliant on fossil fuels. For instance, models trained in coal-powered facilities in Virginia (USA) generate far higher emissions than those trained in regions powered by hydroelectric or nuclear energy. Hardware selection also plays a crucial role; training on energy-efficient tensor processing units (TPUs) can significantly reduce emissions compared to using traditional graphics processing units (GPUs).

Table 17.2 summarizes the estimated carbon emissions associated with training various AI models, illustrating the correlation between model complexity and environmental impact.

Table 17.2: Estimated carbon emissions associated with training various AI models, based on computational requirements and energy consumption. Source: Adapted from (D. Patterson, Gonzalez, Holzle, et al. 2022; Strubell, Ganesh, and McCallum 2019b).

AI Model	Training Compute (FLOPs)	Estimated CO_2 Emissions (kg)	Equivalent Car Miles Driven
GPT-3	3.1×10^{23}	502,000 kg	1.2 million miles
T5-11B	2.3×10^{22}	85,000 kg	210,000 miles
BERT (Base)	3.3×10^{18}	650 kg	1,500 miles
ResNet-50	2.0×10^{17}	35 kg	80 miles

Addressing the sustainability challenges of the design phase requires innovations in training efficiency and computational resource management. Researchers have explored techniques such as sparse training, low-precision arithmetic, and weight-sharing methods to reduce the number of required computations without sacrificing model performance. The use of pre-trained models has also gained traction as a means of minimizing resource consumption. Instead of training models from scratch, researchers can fine-tune smaller versions of pre-trained networks, leveraging existing knowledge to achieve similar results with lower computational costs.

Optimizing model search algorithms further contributes to sustainability. Traditional neural architecture search methods require evaluating a large number of candidate architectures, but recent advances in energy-aware NAS approaches prioritize efficiency by reducing the number of training iterations needed to identify optimal configurations. Companies have also begun implementing carbon-aware computing strategies by scheduling training jobs during periods of lower grid carbon intensity or shifting workloads to data centers with cleaner energy sources (U. Gupta, Elgamal, et al. 2022).

The design phase sets the foundation for the entire AI lifecycle, influencing energy demands in both the training and inference stages. As AI models grow in complexity, their development processes must be reevaluated to ensure that sustainability considerations are integrated at every stage. The decisions made during model design not only determine computational efficiency but also shape the long-term environmental footprint of AI technologies.

17.5.2 Manufacturing Phase

The manufacturing phase of AI systems is one of the most resource-intensive aspects of their lifecycle, involving the fabrication of specialized semiconductor hardware such as GPUs, TPUs, FPGAs, and other AI accelerators. The production of these chips requires large-scale industrial processes, including raw material extraction, wafer fabrication, lithography, doping, and packaging—all of which contribute significantly to environmental impact (@ Bhamra et al. 2024). This phase not only involves high energy consumption but also generates hazardous waste, relies on scarce materials, and has long-term consequences for resource depletion.

17.5.2.1 Fabrication Materials

The foundation of AI hardware lies in semiconductors, primarily silicon-based integrated circuits that power AI accelerators. However, modern AI chips rely on more than just silicon; they require specialty materials such as gallium, indium, arsenic, and helium, each of which carries unique environmental extraction costs. These materials are often classified as critical elements due to their scarcity, geopolitical sensitivity, and high energy costs associated with mining and refining (Bhamra et al. 2024).

Silicon itself is abundant, but refining it into high-purity wafers requires extensive energy-intensive processes. The production of a single 300mm silicon wafer requires over 8,300 liters of water, along with strong acids such as hydrofluoric acid, sulfuric acid, and nitric acid used for etching and cleaning (Cope 2009). The demand for ultra-pure water in semiconductor fabrication places a significant burden on local water supplies, with leading fabs consuming millions of gallons per day.

Beyond silicon, gallium and indium are essential for high-performance compound semiconductors, such as those used in high-speed AI accelerators and 5G communications. The U.S. Geological Survey has classified indium as a critically endangered material, with global supplies estimated to last fewer than 15 years at current consumption rates (M. Davies 2011). Meanwhile, helium, a crucial cooling agent in chip production, is a non-renewable resource that, once released, escapes Earth's gravity, making it permanently unrecoverable. The continued expansion of AI hardware manufacturing is accelerating the depletion of these critical elements, raising concerns about long-term sustainability.

The environmental burden of semiconductor fabrication is further amplified by the use of EUV lithography, a process required for manufacturing sub-5nm chips. EUV systems consume massive amounts of energy, requiring high-powered lasers and complex optics. The International Semiconductor Roadmap estimates that each EUV tool consumes approximately one megawatt (MW) of electricity, significantly increasing the carbon footprint of cutting-edge chip production.

17.5.2.2 Manufacturing Energy Consumption

The energy required to manufacture AI hardware is substantial, with the total energy cost per chip often exceeding its entire operational lifetime energy use. The manufacturing of a single AI accelerator can emit more carbon than years of continuous use in a data center, making fabrication a key hotspot in AI's environmental impact.

17.5.2.3 Hazardous Waste and Water Usage in Fabs

Semiconductor fabrication also generates large volumes of hazardous waste, including gaseous emissions, VOCs, chemical wastewater, and solid byproducts. The acids and solvents used in chip production produce toxic waste streams that require specialized handling to prevent contamination of surrounding ecosystems. Despite advancements in wastewater treatment, trace amounts of metals and chemical residues can still be released into rivers and lakes, affecting aquatic biodiversity and human health (Prakash et al. 2023).

The demand for water in semiconductor fabs has also raised concerns about regional water stress. The TSMC[5] fab in Arizona is projected to consume 8.9 million gallons per day, a figure that accounts for nearly 3% of the city's water supply. While some fabs have begun investing in water recycling systems, these efforts remain insufficient to offset the growing demand.

17.5.2.4 Sustainable Initiatives

Recognizing the sustainability challenges of semiconductor manufacturing, industry leaders have started implementing initiatives to reduce energy consumption, waste generation, and emissions. Companies like Intel, TSMC, and Samsung have pledged to transition towards carbon-neutral semiconductor fabrication through several key approaches. Many fabs are incorporating renewable energy sources, with facilities in Taiwan and Europe increasingly powered by hydroelectric and wind energy. Water conservation efforts have expanded through closed-loop recycling systems that reduce dependence on local water supplies. Manufacturing processes are being redesigned with eco-friendly etching and lithography techniques that minimize hazardous waste generation. Additionally, companies are developing energy-efficient chip architectures, such as low-power AI accelerators optimized for performance per watt, to reduce the environmental impact of both manufacturing and operation. Despite these efforts, the overall environmental footprint of AI chip manufacturing continues to grow as demand for AI accelerators escalates. Without significant improvements in material efficiency, recycling, and fabrication techniques, the manufacturing phase will remain a major contributor to AI's sustainability challenges.

The manufacturing phase of AI hardware represents one of the most resource-intensive and environmentally impactful aspects of AI's lifecycle. The extraction of critical materials, high-energy fabrication processes, and hazardous waste generation all contribute to AI's growing carbon footprint. While industry efforts toward sustainable semiconductor manufacturing are gaining momentum, scaling these initiatives to meet rising AI demand remains a significant challenge.

Addressing the sustainability of AI hardware will require a combination of material innovation, supply chain transparency, and greater investment in circular economy models that emphasize chip recycling and reuse. As AI systems continue to advance, their long-term viability will depend not only on computational efficiency but also on reducing the environmental burden of their underlying hardware infrastructure.

17.5.3 Use Phase

The use phase of AI systems represents one of the most energy-intensive stages in their lifecycle, encompassing both training and inference workloads. As AI adoption grows across industries, the computational requirements for developing and deploying models continue to increase, leading to greater energy consumption and carbon emissions. The operational costs of AI systems extend beyond the direct electricity used in processing; they also include the power demands of data centers, cooling infrastructure, and networking equipment that

[5] Taiwan Semiconductor Manufacturing Company (TSMC) is one of the world's largest semiconductor fabs, consuming millions of gallons of water daily in chip production, raising concerns about water scarcity.

support large-scale AI workloads. Understanding the sustainability challenges of this phase is critical for mitigating AI's long-term environmental impact.

AI model training is among the most computationally expensive activities in the use phase. Training large-scale models involves running billions or even trillions of mathematical operations across specialized hardware, such as GPUs and TPUs, for extended periods. The energy consumption of training has risen sharply in recent years as AI models have grown in complexity. OpenAI's GPT-3, for example, required approximately 1,300 megawatt-hours (MWh) of electricity, an amount equivalent to powering 1,450 U.S. homes for a month (Maslej et al. 2023). The carbon footprint of such training runs depends largely on the energy mix of the data center where they are performed. A model trained in a region relying primarily on fossil fuels, such as coal-powered data centers in Virginia, generates significantly higher emissions than one trained in a facility powered by hydroelectric or nuclear energy.

Beyond training, the energy demands of AI do not end once a model is developed. The inference phase, where a trained model is used to generate predictions, is responsible for an increasingly large share of AI's operational carbon footprint. In real-world applications, inference workloads run continuously, handling billions of requests daily across services such as search engines, recommendation systems, language models, and autonomous systems. The cumulative energy impact of inference is substantial, especially in large-scale deployments. While a single training run for a model like GPT-3 is energy-intensive, inference workloads running across millions of users can consume even more power over time. Studies have shown that inference now accounts for more than 60% of total AI-related energy consumption, exceeding the carbon footprint of training in many cases (D. Patterson, Gonzalez, Holzle, et al. 2022).

Data centers play a central role in enabling AI, housing the computational infrastructure required for training and inference. These facilities rely on thousands of high-performance servers, each drawing significant power to process AI workloads. The power usage effectiveness of a data center, which measures the efficiency of its energy use, directly influences AI's carbon footprint. Many modern data centers operate with PUE values between 1.1 and 1.5, meaning that for every unit of power used for computation, an additional 10% to 50% is consumed for cooling, power conversion, and infrastructure overhead (Barroso, Hölzle, and Ranganathan 2019). Cooling systems, in particular, are a major contributor to data center energy consumption, as AI accelerators generate substantial heat during operation.

The geographic location of data centers has a direct impact on their sustainability. Facilities situated in regions with renewable energy availability can significantly reduce emissions compared to those reliant on fossil fuel-based grids. Companies such as Google and Microsoft have invested in carbon-aware computing strategies, scheduling AI workloads during periods of high renewable energy production to minimize their carbon impact (U. Gupta, Elgamal, et al. 2022). Google's DeepMind, for instance, developed an AI-powered cooling optimization system that reduced data center cooling energy consumption by 40%, lowering the overall carbon footprint of AI infrastructure.

The increasing energy demands of AI raise concerns about grid capacity and sustainability trade-offs. AI workloads often compete with other high-energy sectors, such as manufacturing and transportation, for limited electricity supply. In some regions, the rise of AI-driven data centers has led to increased stress on power grids, necessitating new infrastructure investments. The so-called "duck curve" problem, where renewable energy generation fluctuates throughout the day, poses additional challenges for balancing AI's energy demands with grid availability. The shift toward distributed AI computing and edge processing is emerging as a potential solution to reduce reliance on centralized data centers, shifting some computational tasks closer to end users.

Mitigating the environmental impact of AI's use phase requires a combination of hardware, software, and infrastructure-level optimizations. Advances in energy-efficient chip architectures, such as low-power AI accelerators and specialized inference hardware, have shown promise in reducing per-query energy consumption. AI models themselves are being optimized for efficiency through techniques such as quantization, pruning, and distillation, which allow for smaller, faster models that maintain high accuracy while requiring fewer computational resources. Meanwhile, ongoing improvements in cooling efficiency, renewable energy integration, and data center operations are essential for ensuring that AI's growing footprint remains sustainable in the long term.

As AI adoption continues to expand, energy efficiency must become a central consideration in model deployment strategies. The use phase will remain a dominant contributor to AI's environmental footprint, and without significant intervention, the sector's electricity consumption could grow exponentially. Sustainable AI development requires a coordinated effort across industry, academia, and policymakers to promote responsible AI deployment while ensuring that technological advancements do not come at the expense of long-term environmental sustainability.

17.5.4 Disposal Phase

The disposal phase of AI systems is often overlooked in discussions of sustainability, yet it presents significant environmental challenges. The rapid advancement of AI hardware has led to shorter hardware lifespans, contributing to growing electronic waste (e-waste) and resource depletion. As AI accelerators, GPUs, and high-performance processors become obsolete within a few years, managing their disposal has become a pressing sustainability concern. Unlike traditional computing devices, AI hardware contains complex materials, rare earth elements, and hazardous substances that complicate recycling and waste management efforts. Without effective strategies for repurposing, recycling, or safely disposing of AI hardware, the environmental burden of AI infrastructure will continue to escalate.

The lifespan of AI hardware is relatively short, particularly in data centers where performance efficiency dictates frequent upgrades. On average, GPUs, TPUs, and AI accelerators are replaced every three to five years, as newer, more powerful models enter the market. This rapid turnover results in a constant cycle of hardware disposal, with large-scale AI deployments generating substantial e-waste. Unlike consumer electronics, which may have secondary markets for resale or reuse, AI accelerators often become unviable for commercial use once they are no longer state-of-the-art. The push for ever-faster and more efficient

AI models accelerates this cycle, leading to an increasing volume of discarded high-performance computing hardware.

One of the primary environmental concerns with AI hardware disposal is the presence of hazardous materials. AI accelerators contain heavy metals such as lead, cadmium, and mercury, as well as toxic chemical compounds used in semiconductor fabrication. If not properly handled, these materials can leach into soil and water sources, causing long-term environmental and health hazards. The burning of e-waste releases toxic fumes, contributing to air pollution and exposing workers in informal recycling operations to harmful substances. Studies estimate that only 17.4% of global e-waste is properly collected and recycled, leaving the majority to end up in landfills or informal waste processing sites with inadequate environmental protections (Singh and Ogunseitan 2022).

The complex composition of AI hardware presents significant challenges for recycling. Unlike traditional computing components, which are relatively straightforward to dismantle, AI accelerators incorporate specialized multi-layered circuits, exotic metal alloys, and tightly integrated memory architectures that make material recovery difficult. The disassembly and separation of valuable elements such as gold, palladium, and rare earth metals require advanced recycling technologies that are not widely available. The presence of mixed materials further complicates the process, as some components are chemically bonded or embedded in ways that make extraction inefficient.

Despite these challenges, efforts are being made to develop sustainable disposal solutions for AI hardware. Some manufacturers have begun designing AI accelerators with modular architectures, allowing for easier component replacement and extending the usable lifespan of devices. Research is also underway to improve material recovery processes, making it possible to extract and reuse critical elements such as gallium, indium, and tungsten from discarded chips. Emerging techniques such as hydrometallurgical and biometallurgical processing show promise in extracting rare metals with lower environmental impact compared to traditional smelting and refining methods.

The circular economy model offers a promising approach to mitigating the e-waste crisis associated with AI hardware. Instead of following a linear "use and discard" model, circular economy principles emphasize reuse, refurbishment, and recycling to extend the lifecycle of computing devices. Companies such as Google and Microsoft have launched initiatives to repurpose decommissioned AI hardware for secondary applications, such as running lower-priority machine learning tasks or redistributing functional components to research institutions. These efforts help reduce the overall demand for new semiconductor production while minimizing waste generation.

In addition to corporate sustainability initiatives, policy interventions and regulatory frameworks are critical in addressing the disposal phase of AI systems. Governments worldwide are beginning to implement extended producer responsibility (EPR) policies, which require technology manufacturers to take accountability for the environmental impact of their products throughout their entire lifecycle. In regions such as the European Union, strict e-waste management regulations mandate that electronic manufacturers participate in certified recycling programs and ensure the safe disposal of hazardous materials. However, enforcement remains inconsistent, and significant gaps exist in global e-waste tracking and management.

The future of AI hardware disposal will depend on advancements in recycling technology, regulatory enforcement, and industry-wide adoption of sustainable design principles. The growing urgency of AI-driven e-waste underscores the need for integrated lifecycle management strategies that account for the full environmental impact of AI infrastructure, from raw material extraction to end-of-life recovery. Without concerted efforts to improve hardware sustainability, the rapid expansion of AI will continue to exert pressure on global resources and waste management systems.

17.6 Mitigating Environmental Impact

The rapid expansion of AI has brought remarkable advancements in automation, language understanding, and decision-making, but it has also led to a significant and growing environmental impact. AI models, particularly large-scale deep learning systems, require massive computational resources for both training and inference. This results in high energy consumption, extensive carbon emissions, and resource-intensive hardware manufacturing. As AI adoption accelerates, the challenge of ensuring environmentally sustainable AI development becomes more urgent.

Addressing AI's environmental footprint requires a multi-faceted approach, integrating energy-efficient AI models, optimized hardware, sustainable data center operations, and carbon-aware computing strategies. Additionally, AI systems must be designed with lifecycle sustainability in mind, ensuring that models remain efficient throughout their deployment, from training to inference.

A fundamental principle that must guide all efforts to mitigate AI's environmental impact is Jevon's Paradox. This paradox, observed by William Stanley Jevons in the 19th century[6] (Jevons 1865), says that improvements in technological efficiency can lead to an increase in overall consumption. In the context of AI, even as we develop more energy-efficient models and hardware, the increased accessibility and adoption of AI technologies could lead to a net increase in energy consumption and resource utilization. Therefore, we must approach mitigation strategies with a keen awareness of this potential rebound effect, ensuring that efficiency gains do not inadvertently drive greater consumption. This section explores key strategies for mitigating AI's environmental impact, beginning with sustainable AI development principles.

This effect is illustrated in Figure 17.12. As AI systems become more efficient, the cost per unit of computation decreases, whether for language model tokens, computer vision inferences, or recommendation system predictions. In the figure, moving from point A to point B represents a drop in computation cost. However, this price reduction leads to increased usage across all AI applications, as shown by the corresponding shift from point C to point D on the horizontal axis. While there are savings from reduced costs, the total consumption of AI services increases even more rapidly, ultimately resulting in higher overall resource usage and environmental impact. This dynamic

[6] William Stanley Jevons observed during the Industrial Revolution that technological improvements in coal efficiency paradoxically led to increased coal consumption rather than conservation. In his 1865 book "The Coal Question", he noted that more efficient steam engines made coal power more economical, which expanded its industrial applications and ultimately increased total coal usage. This principle became known as Jevons Paradox.

highlights the core of Jevon's Paradox in AI: efficiency alone is not sufficient to guarantee sustainability.

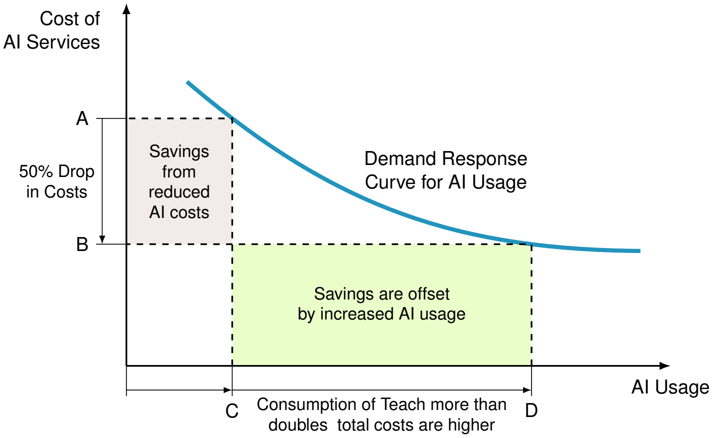


Figure 17.12: Jevon's Paradox in AI efficiency improvements and usage.

17.6.1 Sustainable Development

The design and development of AI models have historically prioritized performance, often at the expense of efficiency. However, as computational demands rise and AI systems scale, this approach is becoming increasingly unsustainable. A single training run of a large transformer-based model can emit as much carbon as five cars over their entire lifetime (Strubell, Ganesh, and McCallum 2019b). Furthermore, many AI models require frequent retraining to adapt to evolving data, compounding their energy consumption. Addressing these sustainability challenges requires a shift from brute-force computation to efficiency-driven innovation. By optimizing model architectures, reducing redundant training, and integrating sustainability principles throughout the AI lifecycle, the environmental impact of AI can be significantly reduced.

17.6.1.1 Energy-Efficient Design

Many deep learning models rely on billions of parameters, requiring trillions of floating-point operations per second (FLOPS) during training and inference. While these large models achieve state-of-the-art performance, research indicates that much of their computational complexity is unnecessary. Many parameters contribute little to final predictions, leading to wasteful resource utilization. To mitigate this inefficiency, several optimization techniques have been developed to reduce the computational overhead of AI models while maintaining accuracy.

One of the most widely used methods for improving energy efficiency is pruning, a technique that removes unnecessary connections from a trained neural network. By systematically eliminating redundant weights, pruning reduces both the model size and the number of computations required during inference. Studies have shown that structured pruning can remove up to 90%

of weights in models such as ResNet-50 while maintaining comparable accuracy. This approach enables AI models to operate efficiently on lower-power hardware, making them more suitable for deployment in resource-constrained environments.

Another technique for reducing energy consumption is quantization, which lowers the numerical precision of computations in AI models. Standard deep learning models typically use 32-bit floating-point precision, but many operations can be performed with 8-bit or even 4-bit integers without significant accuracy loss. By using lower precision, quantization reduces memory requirements, speeds up inference, and lowers power consumption. For example, NVIDIA's TensorRT framework applies post-training quantization to deep learning models, achieving a threefold increase in inference speed while maintaining nearly identical accuracy. Similarly, Intel's Q8BERT demonstrates that quantizing the BERT language model to 8-bit integers can reduce its size by a factor of four with minimal performance degradation (Zafrir et al. 2019).

A third approach, knowledge distillation, allows large AI models to transfer their learned knowledge to smaller, more efficient models. In this process, a large teacher model trains a smaller student model to approximate its predictions, enabling the student model to achieve competitive performance with significantly fewer parameters. Google's DistilBERT exemplifies this technique, retaining 97% of the original BERT model's accuracy while using only 40% of its parameters. Knowledge distillation techniques enable AI practitioners to deploy lightweight models that require substantially less computational power while delivering high-quality predictions.

While these optimization techniques improve efficiency, they also introduce trade-offs. Pruning and quantization can lead to small reductions in model accuracy, requiring fine-tuning to balance performance and sustainability. Knowledge distillation, on the other hand, demands additional training cycles, meaning that energy savings are realized primarily during deployment rather than in the training phase. Furthermore, we must consider Jevon's Paradox: will these efficiency gains lead to a proliferation of AI applications, ultimately increasing overall energy consumption? To counteract this, strategies that combine efficiency with limitations on resource usage are necessary. Nonetheless, these techniques represent essential strategies for reducing the energy footprint of AI models without compromising their effectiveness.

17.6.1.2 Lifecycle-Aware Systems

In addition to optimizing individual models, AI systems must be designed with a broader lifecycle-aware perspective. Many AI deployments operate with a short-term mindset, where models are trained, deployed, and then discarded within a few months. This frequent retraining cycle leads to excessive computational waste. By incorporating sustainability considerations into the AI development pipeline, it is possible to extend model lifespan, reduce unnecessary computation, and minimize environmental impact.

One of the most effective ways to reduce redundant computation is to limit the frequency of full model retraining. Many production AI systems do not require complete retraining from scratch; instead, they can be updated using incremental learning techniques that adapt existing models to new data. Transfer learning is a widely used approach in which a pre-trained model is fine-tuned on a new dataset, significantly reducing the computational cost compared to training a model from the ground up (Narang et al. 2021). This technique is particularly valuable for domain adaptation, where models trained on large general datasets can be customized for specific applications with minimal retraining.

Another critical aspect of lifecycle-aware AI development is the integration of LCA methodologies. LCA provides a systematic framework for quantifying the environmental impact of AI systems at every stage of their lifecycle, from initial training to long-term deployment. Organizations such as MLCommons are actively developing sustainability benchmarks that measure factors such as energy efficiency per inference and carbon emissions per model training cycle (Henderson et al. 2020b). By embedding LCA principles into AI workflows, developers can identify sustainability bottlenecks early in the design process and implement corrective measures before models enter production.

Beyond training efficiency and design evaluation, AI deployment strategies can further enhance sustainability. Cloud-based AI models often rely on centralized data centers, requiring significant energy for data transfer and inference. In contrast, edge computing enables AI models to run directly on end-user devices, reducing the need for constant cloud communication. Deploying AI models on specialized low-power hardware at the edge not only improves latency and privacy but also significantly decreases energy consumption (X. Xu et al. 2021).

However, Jevon's Paradox reminds us that optimizing individual stages might not lead to overall sustainability. For example, even if we improve the recyclability of AI hardware, increased production due to greater demand could still lead to increased resource depletion. Therefore, limiting the production of unneeded hardware is also important. By adopting a lifecycle-aware approach to AI development, practitioners can reduce the environmental impact of AI systems while promoting long-term sustainability.

17.6.1.3 Policy and Incentives

While technical optimizations play a crucial role in mitigating AI's environmental impact, they must be reinforced by policy incentives and industry-wide commitments to sustainability. Several emerging initiatives aim to integrate sustainability principles into AI development at scale.

One promising approach is carbon-aware AI scheduling, where AI workloads are dynamically allocated based on the availability of renewable energy. Companies such as Google have developed scheduling algorithms that shift AI training jobs to times when wind or solar power is abundant, reducing reliance on fossil fuels (D. Patterson, Gonzalez, Le, et al. 2022). These strategies are particularly effective in large-scale data centers, where peak energy demand can be aligned with periods of low-carbon electricity generation.

Benchmarks and leaderboards focused on sustainability are also gaining traction within the AI community. The ML.ENERGY Leaderboard, for example, ranks AI models based on energy efficiency and carbon footprint, encouraging researchers to optimize models not only for performance but also for sustainability. Similarly, MLCommons is working on standardized benchmarks that evaluate AI efficiency in terms of power consumption per inference, providing a transparent framework for comparing the environmental impact of different models.

Regulatory efforts are beginning to shape the future of sustainable AI. The European Union's Sustainable Digital Markets Act[7] has introduced guidelines for transparent AI energy reporting, requiring tech companies to disclose the carbon footprint of their AI operations. As regulatory frameworks evolve, organizations will face increasing pressure to integrate sustainability considerations into their AI development practices (Commission 2023).

By aligning technical optimizations with industry incentives and policy regulations, AI practitioners can ensure that sustainability becomes an integral component of AI development. The shift toward energy-efficient models, lifecycle-aware design, and transparent environmental reporting will be critical in mitigating AI's ecological impact while continuing to drive innovation.

17.6.2 Infrastructure Optimization

The sustainability of AI systems is shaped not only by the efficiency of machine learning models but also by the infrastructure that powers them. While algorithmic improvements such as pruning, quantization, and knowledge distillation reduce computational requirements at the model level, the broader energy footprint of AI is largely dictated by how and where these computations are performed. Large-scale AI workloads are executed in cloud data centers, which house thousands of interconnected servers running continuously to support machine learning training and inference. These facilities consume enormous amounts of electricity, with some hyperscale data centers drawing over 100 megawatts of power, an amount comparable to the energy demand of a small city (N. P. Jones, Johnson, and Montgomery 2021). In addition to direct energy consumption, the cooling requirements of AI infrastructure introduce further sustainability challenges. Data centers must dissipate significant amounts of heat generated by AI accelerators, often relying on energy-intensive air conditioning or water-based cooling systems. As AI adoption continues to expand, these infrastructure-level considerations will play an increasingly central role in determining the overall environmental impact of machine learning systems.

Addressing these challenges requires a shift toward energy-efficient AI infrastructure. The integration of renewable energy into cloud data centers, the adoption of advanced cooling strategies, and the development of carbon-aware workload scheduling can significantly reduce the carbon footprint of AI operations. By designing AI infrastructure to align with sustainability principles, it is possible to minimize the environmental cost of computation while maintaining the performance required for modern machine learning workloads. This section explores the key approaches to optimizing AI infrastructure, focusing on energy-efficient data centers, dynamic workload scheduling based on carbon intensity, and AI-driven cooling strategies. These advancements offer a pathway toward more sustainable AI deployment, ensuring that the growth of machine learning does not come at the expense of long-term environmental responsibility.

[7] Sustainable Digital Markets Act (SDMA): A legislative initiative by the European Union aimed at promoting transparency in AI energy consumption and enforcing sustainability standards in digital markets.

17.6.2.1 Green Data Centers

The increasing computational demands of AI have made data centers one of the largest consumers of electricity in the digital economy. Large-scale cloud data centers provide the infrastructure necessary for training and deploying machine learning models, but their energy consumption is substantial. A single hyperscale data center can consume over 100 megawatts of power, a level comparable to the electricity usage of a small city. Without intervention, the continued growth of AI workloads threatens to push the energy consumption of data centers beyond sustainable levels. The industry must adopt strategies to optimize power efficiency, integrate renewable energy sources, and improve cooling mechanisms to mitigate the environmental impact of AI infrastructure.

One of the most promising approaches to reducing data center emissions is the transition to renewable energy. Major cloud providers, including Google, Microsoft, and Amazon Web Services, have committed to powering their data centers with renewable energy, but implementation challenges remain. Unlike fossil fuel plants, which provide consistent electricity output, renewable sources such as wind and solar are intermittent, with generation levels fluctuating throughout the day. To address this variability, AI infrastructure must incorporate energy storage solutions, such as large-scale battery deployments, and implement intelligent scheduling mechanisms that shift AI workloads to times when renewable energy availability is highest. Google, for example, has set a goal to operate its data centers on 24/7 carbon-free energy by 2030, ensuring that every unit of electricity consumed is matched with renewable generation rather than relying on carbon offsets alone.

Cooling systems represent another major contributor to the energy footprint of data centers, often accounting for 30 to 40 percent of total electricity consumption. Traditional cooling methods rely on air conditioning units and mechanical chillers, both of which require significant power and water resources. To improve efficiency, data centers are adopting alternative cooling strategies that reduce energy waste. Liquid cooling, which transfers heat away from AI accelerators using specially designed coolant systems, is significantly more effective than traditional air cooling and is now being deployed in high-density computing clusters. Free-air cooling, which utilizes natural airflow instead of mechanical refrigeration, has also been adopted in temperate climates, where external conditions allow for passive cooling. Microsoft has taken this a step further by deploying underwater data centers that use the surrounding ocean as a natural cooling mechanism, reducing the need for active temperature regulation.

Beyond hardware-level optimizations, AI itself is being used to improve the energy efficiency of data center operations. DeepMind has developed machine learning algorithms capable of dynamically adjusting cooling parameters based on real-time sensor data. These AI-powered cooling systems analyze temperature, humidity, and fan speeds, making continuous adjustments to optimize energy efficiency. When deployed in Google's data centers, DeepMind's system achieved a 40 percent reduction in cooling energy consumption, demonstrating the potential of AI to enhance the sustainability of the infrastructure that supports machine learning workloads.

However, Jevon's Paradox suggests that even highly efficient data centers could contribute to increased consumption if they enable a massive expansion of AI-driven services. Optimizing the energy efficiency of data centers is critical to reducing the environmental impact of AI, but efficiency alone is not enough. We must also consider strategies for limiting the growth of data center capacity. The integration of renewable energy, the adoption of advanced cooling solutions, and the use of AI-driven optimizations can significantly decrease the carbon footprint of AI infrastructure. As AI continues to scale, these innovations will play a central role in ensuring that machine learning remains aligned with sustainability goals.

17.6.2.2 Carbon-Aware Scheduling

Beyond improvements in hardware and cooling systems, optimizing when and where AI workloads are executed is another critical strategy for reducing AI's environmental impact. The electricity used to power data centers comes from energy grids that fluctuate in carbon intensity based on the mix of power sources available at any given time. Fossil fuel-based power plants supply a significant portion of global electricity, but the share of renewable energy varies by region and time of day. Without optimization, AI workloads may be executed when carbon-intensive energy sources dominate the grid, unnecessarily increasing emissions. By implementing carbon-aware scheduling, AI computations can be dynamically shifted to times and locations where low-carbon energy is available, significantly reducing emissions without sacrificing performance.

Google has pioneered one of the most advanced implementations of carbon-aware computing in its cloud infrastructure. In 2020, the company introduced a scheduling system that delays non-urgent AI tasks until times when renewable energy sources such as solar or wind power are more abundant. This approach enables AI workloads to align with the natural variability of clean energy availability, reducing reliance on fossil fuels while maintaining high computational efficiency. Google has further extended this strategy by geographically distributing AI workloads, moving computations to data centers in regions where clean energy is more accessible. By shifting large-scale AI training jobs from fossil fuel-heavy grids to low-carbon power sources, the company has demonstrated that significant emissions reductions can be achieved through intelligent workload placement.

The potential for carbon-aware scheduling extends beyond hyperscale cloud providers. Companies that rely on AI infrastructure can integrate carbon intensity metrics into their own computing pipelines, making informed decisions about when to run machine learning jobs. Microsoft's sustainability-aware cloud computing initiative allows organizations to select carbon-optimized virtual machines, ensuring that workloads are executed with the lowest possible emissions. Research efforts are also underway to develop open-source carbon-aware scheduling frameworks, enabling a broader range of AI practitioners to incorporate sustainability into their computing strategies.

The effectiveness of carbon-aware AI scheduling depends on accurate real-time data about grid emissions. Electricity providers and sustainability organizations have begun publishing grid carbon intensity data through publicly available APIs, allowing AI systems to dynamically respond to changes in energy supply. For instance, the Electricity Maps API provides real-time CO_2 emissions data for power grids worldwide, enabling AI infrastructure to adjust computational workloads based on carbon availability. As access to grid emissions data improves, carbon-aware computing will become a scalable and widely adoptable solution for reducing the environmental impact of AI operations.

By shifting AI computations to times and places with cleaner energy sources, carbon-aware scheduling represents a powerful tool for making AI infrastructure more sustainable. Unlike hardware-based optimizations that require physical upgrades, scheduling improvements can be implemented through software, offering an immediate and cost-effective pathway to emissions reductions. As more organizations integrate carbon-aware scheduling into their AI workflows, the cumulative impact on reducing global AI-related carbon emissions could be substantial.

While these strategies apply broadly to AI workloads, inference operations present unique sustainability challenges and opportunities. Unlike training, which represents a one-time energy cost, inference constitutes an ongoing and growing energy demand as AI applications scale worldwide. Cloud providers are increasingly adopting carbon-aware scheduling specifically for inference workloads, dynamically shifting these operations to regions powered by abundant renewable energy (Alvim et al. 2022). However, as shown in Figure 17.13, the variability of renewable energy production presents significant challenges. The European grid data illustrates how renewable sources fluctuate throughout the day—solar energy peaks at midday, while wind energy shows distinct peaks in mornings and evenings. Currently, fossil and coal-based generation methods supplement energy needs when renewables fall short.

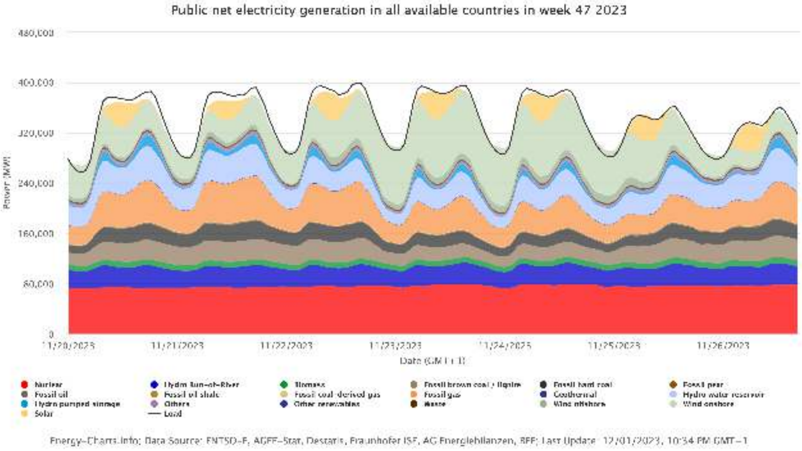


Figure 17.13: Energy sources and generation capabilities. Source: Energy Charts.

To fully leverage carbon-aware scheduling for AI inference workloads, innovation in energy storage solutions is essential for consistent renewable energy use. The base energy load is currently met with nuclear energy—a constant source that produces no direct carbon emissions but lacks the flexibility to accommodate renewable energy variability. Tech companies like Microsoft have shown interest in nuclear energy to power their data centers, as their more constant demand profile (compared to residential use) aligns well with nuclear generation characteristics.

Beyond scheduling, optimizing inference sustainability requires complementary hardware and software innovations. Model quantization techniques enable lower-precision arithmetic to significantly cut power consumption without sacrificing accuracy (A. et al. Gholami 2021). Knowledge distillation methods allow compact, energy-efficient models to replicate the performance of larger, resource-intensive networks (Hinton, Vinyals, and Dean 2015b). Coupled with specialized inference accelerators like Google's TPUs, these approaches substantially reduce inference's environmental impact.

Software frameworks specifically designed for energy efficiency also play a crucial role. Energy-aware AI frameworks, such as Zeus (You, Chung, and Chowdhury 2023) and Perseus (Chung et al. 2023), balance computational speed and power efficiency during both training and inference. These platforms optimize model execution by analyzing trade-offs between speed and energy consumption, facilitating widespread adoption of energy-efficient AI strategies, particularly for inference operations that must run continuously at scale.

17.6.2.3 AI-Driven Thermal Optimization

Cooling systems are one of the most energy-intensive components of AI infrastructure, often accounting for 30-40% of total data center electricity consumption. As AI workloads become more computationally demanding, the heat generated by high-performance accelerators, such as GPUs and TPUs, continues to increase. Without efficient cooling solutions, data centers must rely on power-hungry air conditioning systems or water-intensive thermal management strategies, both of which contribute to AI's overall environmental footprint. To address this challenge, AI-driven cooling optimization has emerged as a powerful strategy for improving energy efficiency while maintaining reliable operations.

DeepMind has demonstrated the potential of AI-driven cooling by deploying machine learning models to optimize temperature control in Google's data centers. Traditional cooling systems rely on fixed control policies, making adjustments based on predefined thresholds for temperature and airflow. However, these rule-based systems often operate inefficiently, consuming more energy than necessary. By contrast, DeepMind's AI-powered cooling system continuously analyzes real-time sensor data, including temperature, humidity, cooling pump speeds, and fan activity, to identify the most energy-efficient configuration for a given workload. Using deep reinforcement learning, the system dynamically adjusts cooling settings to minimize energy consumption while ensuring that computing hardware remains within safe operating temperatures.

When deployed in production, DeepMind's AI-driven cooling system achieved a 40% reduction in cooling energy usage, leading to an overall 15% reduction in total data center power consumption. This level of efficiency improvement demonstrates how AI itself can be used to mitigate the environmental impact of machine learning infrastructure. The success of DeepMind's system has inspired further research into AI-driven cooling, with other cloud providers exploring similar machine learning-based approaches to dynamically optimize thermal management.

Beyond AI-driven control systems, advances in liquid cooling and immersion cooling are further improving the energy efficiency of AI infrastructure. Unlike traditional air cooling, which relies on the circulation of cooled air through server racks, liquid cooling transfers heat directly away from high-performance AI chips using specially designed coolants. This approach significantly reduces the energy required for heat dissipation, allowing data centers to operate at higher densities with lower power consumption. Some facilities have taken this concept even further with immersion cooling, where entire server racks are submerged in non-conductive liquid coolants. This technique eliminates the need for traditional air-based cooling systems entirely, drastically cutting down on electricity usage and water consumption.

Microsoft has also explored innovative cooling solutions, deploying underwater data centers that take advantage of natural ocean currents to dissipate heat. By placing computing infrastructure in sealed submersible enclosures, Microsoft has demonstrated that ocean-based cooling can reduce power usage while extending hardware lifespan due to the controlled and stable underwater environment. While such approaches are still experimental, they highlight the growing interest in alternative cooling technologies that can make AI infrastructure more sustainable.

AI-driven cooling and thermal management represent an immediate and scalable opportunity for reducing the environmental impact of AI infrastructure. Unlike major hardware upgrades, which require capital-intensive investment, software-based cooling optimizations can be deployed rapidly across existing data centers. By leveraging AI to enhance cooling efficiency, in combination with emerging liquid and immersion cooling technologies, the industry can significantly reduce energy consumption, lower operational costs, and contribute to the long-term sustainability of AI systems.

17.6.3 Addressing Full Environmental Footprint

As AI systems continue to scale, efforts to mitigate their environmental impact have largely focused on improving energy efficiency in model design and optimizing data center infrastructure. While these advancements are essential, they address part of the problem. AI's environmental impact extends far beyond operational energy use, encompassing everything from the water consumption in semiconductor manufacturing to the growing burden of electronic waste. A truly sustainable AI ecosystem must account for the full life cycle of AI hardware and software, integrating sustainability at every stage—from material sourcing to disposal.

Earlier in this chapter, we explored the LCA of AI systems, highlighting the substantial carbon emissions, water consumption, and material waste associated with AI hardware manufacturing and deployment. Many of these environmental costs are embedded in the supply chain and do not appear in operational energy reports, leading to an incomplete picture of AI's true sustainability. Moreover, data centers remain water-intensive, with cooling systems consuming millions of gallons per day, and AI accelerators are often refreshed on short life cycles, leading to mounting e-waste.

This section builds on those discussions by examining how AI's broader environmental footprint can be reduced. We explore strategies to mitigate AI's supply chain impact, curb water consumption, and extend hardware longevity. Moving beyond optimizing infrastructure, this approach takes a holistic view of AI sustainability, ensuring that improvements are not just localized to energy efficiency but embedded throughout the entire AI ecosystem.

17.6.3.1 Revisiting Life Cycle Impact

AI's environmental footprint extends far beyond electricity consumption during model training and inference. The full life cycle of AI systems, including hardware manufacturing and disposal, contributes significantly to global carbon emissions, resource depletion, and electronic waste. Earlier in this chapter, we examined the LCA of AI hardware, which revealed that emissions are not solely driven by power consumption but also by the materials and processes involved in fabricating AI accelerators, storage devices, and networking infrastructure.

One of the most striking findings from LCA studies is the embodied carbon cost of AI hardware. Unlike operational emissions, which can be reduced by shifting to cleaner energy sources, embodied emissions result from the raw material extraction, semiconductor fabrication, and supply chain logistics that precede an AI accelerator's deployment. Research indicates that manufacturing emissions alone can account for up to 30% of an AI system's total carbon footprint, with this number potentially growing as data centers improve their reliance on renewable energy sources.

Moreover, AI's water consumption has often been overlooked in sustainability discussions. Semiconductor fabrication plants, in which AI accelerators are produced, are among the most water-intensive industrial facilities in the world, consuming millions of gallons daily for wafer cleaning and chemical processing. Data centers, too, rely on large amounts of water for cooling, with some hyperscale facilities using as much as 450,000 gallons per day—a number that continues to rise as AI workloads become more power-dense. Given that many of the world's chip manufacturing hubs are located in water-stressed regions, such as Taiwan and Arizona, AI's dependence on water raises serious sustainability concerns.

Beyond emissions and water use, AI hardware also contributes to a growing e-waste problem. The rapid evolution of AI accelerators has led to short hardware refresh cycles, where GPUs and TPUs are frequently replaced with newer, more efficient versions. While improving efficiency is critical, discarding functional hardware after only a few years leads to unnecessary electronic waste and resource depletion. Many AI chips contain rare earth metals and toxic components, which, if not properly recycled, can contribute to environmental pollution.

Mitigating AI's environmental impact requires addressing these broader challenges—not just through energy efficiency improvements but by rethinking AI's hardware life cycle, reducing water-intensive processes, and developing sustainable recycling practices. In the following sections, we explore strategies to tackle these issues head-on, ensuring that AI's progress aligns with long-term sustainability goals.

17.6.3.2 Mitigating Supply Chain Impact

Addressing AI's environmental impact requires intervention at the supply chain level, where significant emissions, resource depletion, and waste generation occur before AI hardware even reaches deployment. While much of the discussion around AI sustainability focuses on energy efficiency in data centers, the embodied carbon emissions from semiconductor fabrication, raw material extraction, and hardware transportation represent a substantial and often overlooked portion of AI's total footprint. These supply chain emissions are difficult to offset, making it essential to develop strategies that reduce their impact at the source.

One of the primary concerns is the carbon intensity of semiconductor manufacturing. Fabricating AI accelerators such as GPUs, TPUs, and custom ASICs requires extreme precision and involves processes such as EUV lithography, chemical vapor deposition, and ion implantation, each of which consumes vast amounts of electricity. Since many semiconductor manufacturing hubs operate in regions where grid electricity is still predominantly fossil-fuel-based, the energy demands of chip fabrication contribute significantly to AI's carbon footprint. Research suggests that semiconductor fabrication alone can account for up to 30% of an AI system's total emissions, underscoring the need for more sustainable manufacturing processes.

Beyond carbon emissions, AI's reliance on rare earth elements and critical minerals presents additional sustainability challenges. High-performance AI hardware depends on materials such as gallium, neodymium, and cobalt, which are essential for producing efficient and powerful computing components. However, extracting these materials is highly resource-intensive and often results in toxic waste, deforestation, and habitat destruction. The environmental cost is compounded by geopolitical factors, as over 90% of the world's rare earth refining capacity is controlled by China, creating vulnerabilities in AI's global supply chain. Ensuring responsible sourcing of these materials is critical to reducing AI's ecological and social impact.

Several approaches can mitigate the environmental burden of AI's supply chain. Reducing the energy intensity of chip manufacturing is one avenue, with some semiconductor manufacturers exploring low-energy fabrication processes and renewable-powered production facilities. Another approach focuses on extending the lifespan of AI hardware, as frequent hardware refresh cycles contribute to unnecessary waste. AI accelerators are often designed for peak training performance but remain viable for inference workloads long after they are retired from high-performance computing clusters. Repurposing older AI chips for less computationally intensive tasks, rather than discarding them outright, could significantly reduce the frequency of hardware replacement.

Recycling and closed-loop supply chains also play a crucial role in making AI hardware more sustainable. Recovering and refining valuable materials from retired GPUs, TPUs, and ASICs can reduce reliance on virgin resource extraction while minimizing e-waste. Industry-wide recycling initiatives, combined with hardware design that prioritizes recyclability, could significantly improve AI's long-term sustainability.

Prioritizing supply chain sustainability in AI is not just an environmental necessity but also an opportunity for innovation. By integrating energy-efficient fabrication, responsible material sourcing, and circular hardware design, the AI industry can take meaningful steps toward reducing its environmental impact before these systems ever reach operation. These efforts, combined with continued advances in energy-efficient AI computing, will be essential to ensuring that AI's growth does not come at an unsustainable ecological cost.

17.6.3.3 Reducing Water and Resource Consumption

Mitigating AI's environmental impact requires direct action to reduce its water consumption and resource intensity. AI's reliance on semiconductor fabrication and data centers creates significant strain on water supplies and critical materials, particularly in regions already facing resource scarcity. Unlike carbon emissions, which can be offset through renewable energy, water depletion and material extraction have direct, localized consequences, making it essential to integrate sustainability measures at the design and operational levels.

One of the most effective strategies for reducing AI's water footprint is improving water recycling in semiconductor fabrication. Leading manufacturers are implementing closed-loop water systems, which allow fabs to reuse and treat water rather than continuously consuming fresh supplies. Companies such as Intel and TSMC have already developed advanced filtration and reclamation processes that recover over 80% of the water used in chip production. Expanding these efforts across the industry is essential for minimizing the impact of AI hardware manufacturing.

Similarly, data centers can reduce water consumption by optimizing cooling systems. Many hyperscale facilities still rely on evaporative cooling, which consumes vast amounts of water. Transitioning to direct-to-chip liquid cooling or air-based cooling technologies can significantly reduce water use. In regions with water scarcity, some operators have begun using wastewater or desalinated water for cooling rather than drawing from potable sources. These methods help mitigate the environmental impact of AI infrastructure while maintaining efficient operation.

On the materials side, reducing AI's dependency on rare earth metals and critical minerals is crucial for long-term sustainability. While some materials, such as silicon, are abundant, others, including gallium, neodymium, and cobalt, are subject to geopolitical constraints and environmentally damaging extraction methods. Researchers are actively exploring alternative materials and low-waste manufacturing processes to reduce reliance on these limited resources. Additionally, recycling programs for AI accelerators and other computing hardware can recover valuable materials, reducing the need for virgin extraction.

Beyond individual mitigation efforts, industry-wide collaboration is necessary to develop standards for responsible water use, material sourcing, and recycling programs. Governments and regulatory bodies can also incentivize sustainable practices by enforcing water conservation mandates, responsible mining regulations, and e-waste recycling requirements. By prioritizing these mitigation strategies, the AI industry can work toward minimizing its ecological footprint while continuing to advance technological progress.

17.6.3.4 Systemic Sustainability Approaches

Mitigating AI's environmental impact requires more than isolated optimizations—it demands a systemic shift toward sustainable AI development. Addressing the long-term sustainability of AI means integrating circular economy principles, establishing regulatory policies, and fostering industry-wide collaboration to ensure that sustainability is embedded into the AI ecosystem from the ground up.

Jevon's Paradox highlights the limitations of focusing solely on individual efficiency improvements. We need systemic solutions that address the broader drivers of AI consumption. This includes policies that promote sustainable AI practices, incentives for responsible resource usage, and public awareness campaigns that encourage mindful AI consumption.

One of the most effective ways to achieve lasting sustainability is by aligning AI development with circular economy principles. Unlike the traditional linear model of "build, use, discard," a circular approach prioritizes reuse, refurbishment, and recycling to extend the lifespan of AI hardware (Stahel 2016). Manufacturers and cloud providers can adopt modular hardware designs, allowing individual components, including memory and accelerators, to be upgraded without replacing entire servers. In addition, AI hardware should be designed with recyclability in mind, ensuring that valuable materials can be extracted and reused instead of contributing to electronic waste.

Regulatory frameworks also play a crucial role in enforcing sustainability standards. Governments can introduce carbon transparency mandates, requiring AI infrastructure providers to report the full lifecycle emissions of their operations, including embodied carbon from manufacturing (Masanet et al. 2020b). Additionally, stricter water use regulations for semiconductor fabs and e-waste recycling policies can help mitigate AI's resource consumption. Some jurisdictions have already implemented extended producer responsibility laws, which hold manufacturers accountable for the end-of-life disposal of their products. Expanding these policies to AI hardware could incentivize more sustainable design practices.

At the industry level, collaborative efforts are essential for scaling sustainable AI practices. Leading AI companies and research institutions should establish shared sustainability benchmarks that track energy efficiency, carbon footprint, and resource usage. Furthermore, standardized green AI certifications could guide consumers and enterprises toward more sustainable technology choices (Strubell, Ganesh, and McCallum 2019a). Cloud providers can also commit to 24/7 carbon-free energy (CFE) goals, ensuring that AI workloads are powered by renewable sources in real-time rather than relying on carbon offsets that fail to drive meaningful emissions reductions.

Achieving systemic change in AI sustainability requires a multi-stakeholder approach. Governments, industry leaders, and researchers must work together to set sustainability standards, invest in greener infrastructure, and transition toward a circular AI economy. By embedding sustainability into the entire AI development pipeline, the industry can move beyond incremental optimizations and build a truly sustainable foundation for future innovation.

17.6.4 Case Study: Google's Framework

To mitigate emissions from rapidly expanding AI workloads, Google engineers identified four key optimization areas, identified as the '4 Ms', where systematic improvements collectively reduce the carbon footprint of machine learning:

- **Model**: The selection of efficient AI architectures reduces computation requirements by 5-10X without compromising model quality. Google has extensively researched sparse models and neural architecture search methodologies, resulting in efficient architectures such as the Evolved Transformer and Primer.
- **Machine**: The implementation of AI-specific hardware offers 2-5X improvements in performance per watt compared to general-purpose systems. Google's TPUs demonstrate 5-13X greater carbon efficiency relative to non-optimized GPUs.
- **Mechanization**: The utilization of optimized cloud computing infrastructure with high utilization rates yields 1.4-2X energy reductions compared to conventional on-premise data centers. Google's facilities consistently exceed industry standards for PUE.
- **Map**: The strategic positioning of data centers in regions with low-carbon electricity supplies reduces gross emissions by 5-10X. Google maintains real-time monitoring of renewable energy usage across its global infrastructure.

The combined effect of these practices produces multiplicative efficiency gains. For instance, implementing the optimized Transformer model on TPUs in strategically located data centers reduced energy consumption by a factor of 83 and CO_2 emissions by a factor of 747.

Despite substantial growth in AI deployment across Google's product ecosystem, systematic efficiency improvements have effectively constrained energy consumption growth. A significant indicator of this progress is the observation that AI workloads have maintained a consistent 10% to 15% proportion of Google's total energy consumption from 2019 through 2021. As AI functionality expanded across Google's services, corresponding increases in compute cycles were offset by advancements in algorithms, specialized hardware, infrastructure design, and geographical optimization.

Empirical case studies demonstrate how engineering principles focused on sustainable AI development enable simultaneous improvements in both performance and environmental impact. For example, comparative analysis between GPT-3 (considered state-of-the-art in mid-2020) and Google's GLaM model reveals improved accuracy metrics alongside reduced training computation requirements and lower-carbon energy sources—resulting in a 14-fold reduction in CO_2 emissions within an 18-month development cycle.

Furthermore, Google's analysis indicates that previous published estimates overestimated machine learning's energy requirements by factors ranging from 100 to 100,000X due to methodological limitations and absence of empirical measurements. Through transparent reporting of optimization metrics, Google provides a factual basis for efficiency initiatives while correcting disproportionate projections regarding machine learning's environmental impact.

While substantial progress has been achieved in constraining the carbon footprint of AI operations, Google acknowledges that continued efficiency advancements are essential for responsible innovation as AI applications proliferate. Their ongoing optimization framework encompasses:

1. **Life-Cycle Analysis**: Demonstrating that computational investments such as neural architecture search, while initially resource-intensive, generate significant downstream efficiencies that outweigh initial costs. Despite higher energy expenditure during the discovery phase compared to manual engineering approaches, NAS ultimately reduces cumulative emissions by generating optimized architectures applicable across numerous deployments.
2. **Resource Allocation Prioritization**: Concentrating sustainability initiatives on data center and server-side optimization where energy consumption is most concentrated. While Google continues to enhance inference efficiency on edge devices, primary focus remains on training infrastructure and renewable energy procurement to maximize environmental return on investment.
3. **Economies of Scale**: Leveraging the efficiency advantages inherent in well-designed cloud infrastructure through workload consolidation. As computation transitions from distributed on-premise environments to centralized providers with robust sustainability frameworks, aggregate emissions reductions accelerate.
4. **Renewable Energy Integration**: Prioritizing renewable energy procurement, as Google has achieved a 100% match of energy consumption with renewable sources since 2017, to further reduce the environmental impact of computational workloads.

These integrated approaches indicate that AI efficiency improvements are accelerating rather than plateauing. Google's multifaceted strategy combining systematic measurement, carbon-aware development methodologies, transparency in reporting, and renewable energy transition establishes a replicable framework for sustainable AI scaling. These empirical results provide a foundation for broader industry adoption of comprehensive sustainability practices.

17.7 Embedded AI and E-Waste

The deployment of AI is rapidly expanding beyond centralized data centers into edge and embedded devices, enabling real-time decision-making without requiring constant cloud connectivity. This shift has led to major efficiency gains, reducing latency, bandwidth consumption, and network congestion while enabling new applications in smart consumer devices, industrial automation, healthcare, and autonomous systems. However, the rise of embedded AI brings new environmental challenges, particularly regarding electronic waste, disposable smart devices, and planned obsolescence.

Unlike high-performance AI accelerators in data centers, which are designed for long-term use and high computational throughput, embedded AI hardware is often small, low-cost, and disposable. Many AI-powered IoT sensors, wearables, and smart appliances are built with short lifespans and limited upgradeability, making them difficult, if not entirely impossible, to repair or recycle (C. P. Baldé 2017). As a result, these devices contribute to a rapidly growing electronic waste crisis, one that remains largely overlooked in discussions on AI sustainability.

The scale of this issue is staggering. As illustrated in Figure 17.14, the number of Internet of Things (IoT) devices is projected to exceed 30 billion by 2030, with AI-powered chips increasingly embedded into everything from household appliances and medical implants to industrial monitoring systems and agricultural sensors (Statista 2022). This exponential growth in connected devices presents a significant environmental challenge, as many of these devices will become obsolete within just a few years, leading to an unprecedented surge in e-waste. Without sustainable design practices and improved lifecycle management, the expansion of AI at the edge risks exacerbating global electronic waste accumulation and straining recycling infrastructure.

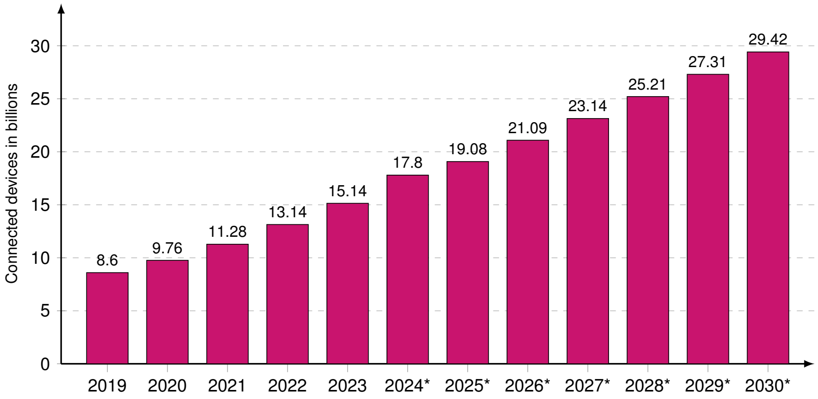


Figure 17.14: Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2023. Source: Statista.

While AI-powered data centers have been scrutinized for their carbon footprint and energy demands, far less attention has been paid to the environmental cost of embedding AI into billions of short-lived devices. Addressing this challenge requires rethinking how AI hardware is designed, manufactured, and disposed of, ensuring that edge AI systems contribute to technological progress without leaving behind an unsustainable legacy of waste.

17.7.1 E-Waste Crisis

Electronic waste, or e-waste, is one of the fastest-growing environmental challenges of the digital age. Defined as discarded electronic devices containing batteries, circuit boards, and semiconductor components, e-waste presents severe risks to both human health and the environment. Toxic materials such as lead, mercury, cadmium, and brominated flame retardants, commonly found in AI-enabled hardware, can contaminate soil and groundwater when improperly disposed of. Despite the potential for recycling and material recovery, most e-waste remains improperly handled, leading to hazardous waste accumulation and significant environmental degradation.

The scale of the problem is staggering. Today, global e-waste production exceeds 50 million metric tons annually, with projections indicating that this figure will surpass 75 million tons by 2030 as consumer electronics and AI-powered IoT devices continue to proliferate. According to the United Nations, e-waste generation could reach 120 million tons per year by 2050 if current consumption patterns persist (Un and Forum 2019). The combination of short product lifespans, rising global demand, and limited recycling infrastructure has accelerated this crisis.

AI-driven consumer devices, such as smart speakers, fitness trackers, and home automation systems, are among the most significant contributors to e-waste. Unlike modular and serviceable computing systems, many of these devices are designed to be disposable, meaning that when a battery fails or a component malfunctions, the entire product is discarded rather than repaired. This built-in disposability exacerbates the unsustainable cycle of consumption and waste, leading to higher material extraction rates and increased pressure on waste management systems.

Developing nations are disproportionately affected by e-waste dumping, as they often lack the infrastructure to process obsolete electronics safely. In 2019, only 13% to 23% of e-waste in lower-income countries was formally collected for recycling, with the remainder either incinerated, illegally dumped, or manually dismantled in unsafe conditions (Un and Forum 2019). Many discarded AI-powered devices end up in informal recycling operations, where low-paid workers are exposed to hazardous materials without proper protective equipment. Open-air burning of plastic components and crude metal extraction methods release toxic fumes and heavy metals into the surrounding environment, posing severe health risks.

The global recycling rate for e-waste remains alarmingly low, with only 20% of all discarded electronics processed through environmentally sound recycling channels. The remaining 80% is either landfilled, incinerated, or dumped illegally, leading to long-term environmental contamination and resource depletion. Without stronger policies, better product design, and expanded e-waste management systems, the rapid growth of AI-powered devices will significantly worsen this crisis.

AI-driven electronics should not become another major contributor to the global e-waste problem. Tackling this challenge requires a multi-pronged approach, including more sustainable design practices, stronger regulatory oversight, and greater investment in global e-waste recycling infrastructure. Without intervention, AI’s environmental impact will extend far beyond its energy consumption, leaving behind a legacy of toxic waste and resource depletion.

17.7.2 Disposable Electronics

The rapid proliferation of low-cost AI-powered microcontrollers, smart sensors, and connected devices has transformed various industries, from consumer electronics and healthcare to industrial automation and agriculture. While these embedded AI systems enable greater efficiency and automation, their short lifespans and non-recyclable designs pose a significant sustainability challenge. Many of these devices are treated as disposable electronics, designed with limited durability, non-replaceable batteries, and little to no repairability, making them destined for the waste stream within just a few years of use.

One of the primary drivers of AI-powered device disposability is the falling cost of microelectronics. The miniaturization of computing hardware has enabled manufacturers to embed tiny AI processors and wireless connectivity modules into everyday products, often for under $1 per chip. As a result, AI functionality is increasingly being integrated into single-use and short-lived products, including smart packaging, connected medical devices, wearables, and home appliances. While these innovations improve convenience and real-time data collection, they lack proper end-of-life management strategies, leading to a surge in hard-to-recycle electronic waste (V. Forti 2020).

17.7.2.1 Non-Replaceable Batteries Cost

Many disposable AI devices incorporate sealed, non-replaceable lithium-ion batteries, making them inherently unsustainable. Smart earbuds, wireless sensors, and even some fitness trackers lose functionality entirely once their batteries degrade, forcing consumers to discard the entire device. Unlike modular electronics with user-serviceable components, most AI-powered wearables and IoT devices are glued or soldered shut, preventing battery replacement or repair.

This issue extends beyond consumer gadgets. Industrial AI sensors and remote monitoring devices, often deployed in agriculture, infrastructure, and environmental monitoring, frequently rely on non-replaceable batteries with a limited lifespan. Once depleted, these sensors, many of which are installed in remote or difficult-to-access locations, become e-waste, requiring costly and environmentally disruptive disposal or replacement (Ciez and Whitacre 2019).

The environmental impact of battery waste is particularly concerning. Lithium mining, essential for battery production, is an energy-intensive process that consumes vast amounts of water and generates harmful byproducts. Additionally, the improper disposal of lithium batteries poses fire and explosion risks, particularly in landfills and waste processing facilities. As the demand for AI-powered devices grows, addressing the battery sustainability crisis will be critical to mitigating AI's long-term environmental footprint.

17.7.2.2 Recycling Challenges

Unlike traditional computing hardware, including desktop computers and enterprise servers, which can be disassembled and refurbished, most AI-enabled consumer electronics are not designed for recycling. Many of these devices contain mixed-material enclosures, embedded circuits, and permanently attached components, making them difficult to dismantle and recover materials from (Patel et al. 2016).

Additionally, AI-powered IoT devices are often too small to be efficiently recycled using conventional e-waste processing methods. Large-scale electronics, such as laptops and smartphones, have well-established recycling programs that allow for material recovery. In contrast, tiny AI-powered sensors, earbuds, and embedded chips are often too costly and labor-intensive to separate into reusable components. As a result, they frequently end up in landfills or incinerators, contributing to pollution and resource depletion.

The environmental impact of battery waste is particularly concerning. Lithium mining, essential for battery production, is an energy-intensive process that consumes vast amounts of water and generates harmful byproducts (Bouri 2015). Additionally, the improper disposal of lithium batteries poses fire and explosion risks, particularly in landfills and waste processing facilities. As the demand for AI-powered devices grows, addressing the battery sustainability crisis will be critical to mitigating AI's long-term environmental footprint (Zhan, Oldenburg, and Pan 2018).

17.7.2.3 Need for Sustainable Design

Addressing the sustainability challenges of disposable AI electronics requires a fundamental shift in design philosophy. Instead of prioritizing cost-cutting and short-term functionality, manufacturers must embed sustainability principles into the development of AI-powered devices. This includes:

- **Designing for longevity**: AI-powered devices should be built with replaceable components, modular designs, and upgradable software to extend their usability.
- **Enabling battery replacement**: Consumer and industrial AI devices should incorporate easily swappable batteries rather than sealed enclosures that prevent repair.
- **Standardizing repairability**: AI hardware should adopt universal standards for repair, ensuring that components can be serviced rather than discarded.
- **Developing biodegradable or recyclable materials**: Research into eco-friendly circuit boards, biodegradable polymers, and sustainable packaging can help mitigate waste.

Incentives and regulations can also encourage manufacturers to prioritize sustainable AI design. Governments and regulatory bodies can implement right-to-repair laws, extended producer responsibility policies, and e-waste take-back programs to ensure that AI-powered devices are disposed of responsibly. Additionally, consumer awareness campaigns can educate users on responsible e-waste disposal and encourage sustainable purchasing decisions.

The future of AI-powered electronics must be circular rather than linear, ensuring that devices are designed with sustainability in mind and do not contribute disproportionately to the global e-waste crisis. By rethinking design, improving recyclability, and promoting responsible disposal, the industry can mitigate the negative environmental impacts of AI at the edge while still enabling technological progress.

17.7.3 AI Hardware Obsolescence

The concept of planned obsolescence refers to the intentional design of products with artificially limited lifespans, forcing consumers to upgrade or replace them sooner than necessary. While this practice has long been associated with consumer electronics and household appliances, it is increasingly prevalent in AI-powered hardware, from smartphones and wearables to industrial AI sensors and cloud infrastructure. This accelerated replacement cycle not only drives higher consumption and production but also contributes significantly to the growing e-waste crisis (Slade 2007).

One of the most visible examples of planned obsolescence in AI hardware is the software-driven degradation of device performance. Many manufacturers introduce software updates that, while ostensibly meant to enhance security and functionality, often degrade the performance of older devices. For example, Apple has faced scrutiny for deliberately slowing down older iPhone models via iOS updates (Luna 2018a). While the company claimed that these updates were meant to prevent battery-related shutdowns, critics argued that they pushed consumers toward unnecessary upgrades rather than encouraging repair or battery replacement.

This pattern extends to AI-powered consumer electronics, where firmware updates can render older models incompatible with newer features, effectively forcing users to replace their devices. Many smart home systems, connected appliances, and AI assistants suffer from forced obsolescence due to discontinued cloud support or software services, rendering hardware unusable even when physically intact (Luna 2018b)[8].

17.7.3.1 Lock-In and Proprietary Components

Another form of planned obsolescence arises from hardware lock-in, where manufacturers deliberately prevent users from repairing or upgrading their devices. Many AI-powered devices feature proprietary components, making it impossible to swap out batteries, upgrade memory, or replace failing parts. Instead of designing for modularity and longevity, manufacturers prioritize sealed enclosures and soldered components, ensuring that even minor failures lead to complete device replacement (Johnson 2018).

For example, many AI wearables and smart devices integrate non-replaceable batteries, meaning that when the battery degrades (often in just two to three years), the entire device becomes e-waste. Similarly, smartphones, laptops, and AI-enabled tablets increasingly use soldered RAM and storage, preventing users from upgrading hardware and extending its lifespan (M. Russell 2022).

Planned obsolescence also affects industrial AI hardware, including AI-powered cameras, factory sensors, and robotics. Many industrial automation systems rely on vendor-locked software ecosystems, where manufacturers discontinue support for older models to push customers toward newer, more expensive replacements. This creates a cycle of forced upgrades, where com-

[8] Apple was fined €25 million by French regulators in 2020 for intentionally slowing down older iPhone models without informing users, a practice that has sparked global debate on software-induced obsolescence.

panies must frequently replace otherwise functional AI hardware simply to maintain software compatibility (Sharma 2020)[9].

17.7.3.2 Environmental Cost

Planned obsolescence is not just a financial burden on consumers—it has severe environmental consequences. By shortening product lifespans and discouraging repairability, manufacturers increase the demand for new electronic components, leading to higher resource extraction, energy consumption, and carbon emissions.

The impact of this cycle is particularly concerning given the high environmental cost of semiconductor manufacturing. Producing AI chips, GPUs, and other advanced computing components requires vast amounts of water, rare earth minerals, and energy[10]. For example, a single 5nm semiconductor fabrication plant consumes millions of gallons of ultrapure water daily and relies on energy-intensive processes that generate significant CO_2 emissions (Mills and Le Hunte 1997; Harris 2023). When AI-powered devices are discarded prematurely, the environmental cost of manufacturing is effectively wasted, amplifying AI's overall sustainability challenges.

Additionally, many discarded AI devices contain hazardous materials, including lead, mercury, and brominated flame retardants, which can leach into the environment if not properly recycled (Puckett 2016). The acceleration of AI-powered consumer electronics and industrial hardware turnover will only worsen the global e-waste crisis, further straining waste management and recycling systems.

17.7.3.3 Extending Hardware Lifespan

Addressing planned obsolescence requires a shift in design philosophy, moving toward repairable, upgradable, and longer-lasting AI hardware. Some potential solutions include:

- **Right-to-Repair Legislation**: Many governments are considering right-to-repair laws, which would require manufacturers to provide repair manuals, replacement parts, and diagnostic tools for AI-powered devices. This would enable consumers and businesses to extend hardware lifespans rather than replacing entire systems (Johnson 2018).
- **Modular AI Hardware**: Designing AI-powered devices with modular components—such as replaceable batteries, upgradeable memory, and standardized ports—can significantly reduce electronic waste while improving cost-effectiveness for consumers (F. C. Inc. 2022).
- **Modular AI Hardware**: Designing AI-powered devices with modular components, such as replaceable batteries, upgradeable memory, and standardized ports, can significantly reduce electronic waste while improving cost-effectiveness for consumers (F. C. Inc. 2022).
- **Extended Software Support**: Companies should commit to longer software support cycles, ensuring that older AI-powered devices remain functional rather than being rendered obsolete due to artificial compatibility constraints (S. Brown 2021).

[9] Many industrial AI systems rely on proprietary software ecosystems, where manufacturers discontinue updates and support for older hardware, forcing companies to purchase new equipment to maintain compatibility.

[10] Semiconductor fabrication is one of the most resource-intensive manufacturing processes, consuming vast amounts of water and energy while generating hazardous chemical waste.

- **Consumer Awareness & Circular Economy**: Encouraging trade-in and recycling programs, along with consumer education on sustainable AI purchasing, can help shift demand toward repairable and long-lasting devices (Cheshire 2021).

Several tech companies are already experimenting with more sustainable AI hardware. For example, Framework, a startup focused on modular laptops, offers fully repairable, upgradeable systems that prioritize long-term usability over disposable design. Similar efforts in the smartphone and AI-driven IoT sectors could help reduce the environmental footprint of planned obsolescence.

The widespread adoption of AI-powered devices presents a critical opportunity to rethink the lifecycle of electronics. If left unchecked, planned obsolescence will continue to drive wasteful consumption patterns, accelerate e-waste accumulation, and exacerbate the resource extraction crisis. However, with policy interventions, industry innovation, and consumer advocacy, AI hardware can be designed for durability, repairability, and sustainability.

The future of AI should not be disposable. Instead, companies, researchers, and policymakers must prioritize long-term sustainability, ensuring that AI's environmental footprint is minimized while its benefits are maximized. Addressing planned obsolescence in AI hardware is a key step toward making AI truly sustainable—not just in terms of energy efficiency but in its entire lifecycle, from design to disposal.

17.8 Policy and Regulation

The increasing energy consumption and carbon emissions of AI systems have raised concerns among policymakers, industry leaders, and environmental advocates. As AI adoption accelerates, regulatory frameworks are becoming essential to ensure that AI development and deployment align with global sustainability goals. Without policy intervention, the rapid scaling of AI infrastructure risks exacerbating climate change, resource depletion, and electronic waste generation (Vinuesa et al. 2020).

Policymakers face a delicate balancing act—on one hand, AI innovation drives economic growth and scientific progress, but on the other, its unchecked expansion could have significant environmental consequences. To address this, policy mechanisms such as measurement and reporting mandates, emission restrictions, financial incentives, and self-regulatory initiatives are being explored worldwide. Government agencies, international organizations, and private sector coalitions are working to establish standardized methodologies for assessing AI's carbon footprint, encourage efficiency improvements, and promote green AI infrastructure investments.

However, policy fragmentation across regions poses challenges. The European Union, for instance, is leading regulatory efforts through initiatives like the AI Act[11] and sustainability disclosure requirements under the Corporate Sustainability Reporting Directive (CSRD)[12], while U.S. policymakers have largely relied on voluntary reporting and market-based incentives. China and other nations are taking their own approaches, creating potential barriers to a unified global AI sustainability strategy

[11] The European Commission's AI Act regulates AI systems' development, deployment, and use, including provisions for sustainability reporting and energy efficiency requirements.

[12] The Corporate Sustainability Reporting Directive (CSRD) mandates that large companies disclose environmental and social information, including AI-related emissions and energy consumption.

This section explores the various policy tools available for mitigating AI's environmental impact, analyzing the role of governments, regulatory bodies, and industry-led efforts. By examining both mandatory and voluntary approaches, we assess how regulations can drive AI sustainability without impeding technological progress.

17.8.1 Measurement and Reporting

A critical first step toward mitigating AI's environmental impact is accurate measurement and transparent reporting of energy consumption and carbon emissions. Without standardized tracking mechanisms, it is difficult to assess AI's true sustainability impact or identify areas for improvement. Government regulations and industry initiatives are beginning to mandate energy audits, emissions disclosures, and standardized efficiency metrics for AI workloads. These policies aim to increase transparency, inform better decision-making, and hold organizations accountable for their environmental footprint.

The lack of universally accepted metrics for assessing AI's environmental impact has been a significant challenge. Current sustainability evaluations often rely on ad hoc reporting by companies, with inconsistent methodologies for measuring energy consumption and emissions. To address this, policymakers and industry leaders are advocating for formalized sustainability benchmarks that assess AI's carbon footprint at multiple levels. Computational complexity and model efficiency are key factors, as they determine how much computation is required for a given AI task. Data center efficiency, often measured through power usage effectiveness, plays a crucial role in evaluating how much of a data center's power consumption directly supports computation rather than being lost to cooling and infrastructure overhead. The carbon intensity of energy supply is another critical consideration, as AI operations running on grids powered primarily by fossil fuels have a far greater environmental impact than those powered by renewable energy sources.

Several industry efforts are working toward standardizing sustainability reporting for AI. The MLCommons benchmarking consortium has begun incorporating energy efficiency as a factor in AI model assessments, recognizing the need for standardized comparisons of model energy consumption. Meanwhile, regulatory bodies are pushing for mandatory disclosures. In Europe, the proposed AI Act includes provisions for requiring organizations using AI at scale to report energy consumption and carbon emissions associated with their models. The European Commission has signaled that sustainability reporting requirements for AI may soon be aligned with broader environmental disclosure regulations under the CSRD.

One of the biggest challenges in implementing AI sustainability reporting is balancing transparency with the potential burden on organizations. While greater transparency is essential for accountability, requiring detailed reporting for every AI workload could create excessive overhead, particularly for smaller firms and research institutions. To address this, policymakers are exploring scalable approaches that integrate sustainability considerations into existing industry standards without imposing rigid compliance costs. Developing lightweight reporting mechanisms that leverage existing monitoring tools within data centers and cloud platforms can help ease this burden while still improving visibility into AI's environmental footprint.

To be most constructive, measurement and reporting policies should focus on enabling continuous refinement rather than imposing simplistic restrictions or rigid caps. Given AI's rapid evolution, regulations that incorporate flexibility while embedding sustainability into evaluation metrics will be most effective in driving meaningful reductions in energy consumption and emissions. Rather than stifling innovation, well-designed policies can encourage AI developers to prioritize efficiency from the outset, fostering a culture of responsible AI design that aligns with long-term sustainability goals.

17.8.2 Restriction Mechanisms

Beyond measurement and reporting mandates, direct policy interventions can restrict AI's environmental impact through regulatory limits on energy consumption, emissions, or model scaling. While AI's rapid growth has spurred innovation, it has also introduced new sustainability challenges that may require governments to impose guardrails to curb excessive environmental costs. Restrictive mechanisms, such as computational caps, conditional access to public resources, financial incentives, and even outright bans on inefficient AI practices, are all potential tools for reducing AI's carbon footprint. However, their effectiveness depends on careful policy design that balances sustainability with continued technological advancement.

One potential restriction mechanism involves setting limits on the computational power available for training large AI models. The European Commission's proposed AI Act has explored this concept by introducing economy-wide constraints on AI training workloads. This approach mirrors emissions trading systems (ETS)[13] in environmental policy, where organizations must either operate within predefined energy budgets or procure additional capacity through regulated exchanges. While such limits could help prevent unnecessary computational waste, they also raise concerns about limiting innovation, particularly for researchers and smaller companies that may struggle to access high-performance computing resources (Schwartz et al. 2020).

Another policy tool involves conditioning access to public datasets and government-funded computing infrastructure based on model efficiency. AI researchers and developers increasingly rely on large-scale public datasets and subsidized cloud resources to train models. Some have proposed that governments could restrict these resources to AI projects that meet strict energy efficiency criteria. For instance, the MLCommons benchmarking consortium could integrate sustainability metrics into its standardized performance leaderboards, incentivizing organizations to optimize for efficiency alongside accuracy. However, while conditioned access could promote sustainable AI practices, it also risks creating disparities by limiting access to computational resources for those unable to meet predefined efficiency thresholds.

Financial incentives and disincentives represent another regulatory mechanism for driving sustainable AI. Carbon taxes on AI-related compute consumption could discourage excessive model scaling while generating funds for efficiency-focused research. Similar to existing environmental regulations,

[13] Emissions trading systems (ETS) are market-based mechanisms that cap the total amount of greenhouse gas emissions allowed within a jurisdiction, with organizations required to purchase or trade emissions allowances to meet their compliance obligations.

organizations could be required to pay fees based on the emissions associated with their AI workloads, encouraging them to optimize for lower energy consumption. Conversely, tax credits could reward companies developing efficient AI techniques, fostering investment in greener computing technologies. While financial mechanisms can effectively guide market behavior, they must be carefully calibrated to avoid disproportionately burdening smaller AI developers or discouraging productive use cases.

In extreme cases, outright bans on particularly wasteful AI applications may be considered. If measurement data consistently pinpoints certain AI practices as disproportionately harmful with no feasible path to remediation, governments may choose to prohibit these activities altogether. However, defining harmful AI use cases is challenging due to AI's dual-use nature, where the same technology can have both beneficial and detrimental applications. Policymakers must approach bans cautiously, ensuring that restrictions target clearly unsustainable practices without stifling broader AI innovation.

Ultimately, restriction mechanisms must strike a careful balance between environmental responsibility and economic growth. Well-designed policies should encourage AI efficiency while preserving the flexibility needed for continued technological progress. By integrating restrictions with incentives and reporting mandates, policymakers can create a comprehensive framework for guiding AI toward a more sustainable future.

17.8.3 Government Incentives

In addition to regulatory restrictions, governments can play a proactive role in advancing sustainable AI development through incentives that encourage energy-efficient practices. Financial support, tax benefits, grants, and strategic investments in Green AI research can drive the adoption of environmentally friendly AI technologies. Unlike punitive restrictions, incentives provide positive reinforcement, making sustainability a competitive advantage rather than a regulatory burden.

One common approach to promoting sustainability is through tax incentives. Governments already offer tax credits for adopting renewable energy sources, such as the U.S. Residential Clean Energy Credit and commercial energy efficiency deductions. Similar programs could be extended to AI companies that optimize their models and infrastructure for lower energy consumption. AI developers who integrate efficiency-enhancing techniques, such as model pruning, quantization, or adaptive scheduling, could qualify for tax reductions, creating a financial incentive for Green AI development.

Beyond tax incentives, direct government funding for sustainable AI research is an emerging strategy. Spain has already committed 300 million euros toward AI projects that explicitly focus on sustainability. Such funding can accelerate breakthroughs in energy-efficient AI by supporting research into novel low-power algorithms, specialized AI hardware, and eco-friendly data center designs. Public-private partnerships can further enhance these efforts, allowing AI companies to collaborate with research institutions and government agencies to pioneer sustainable solutions.

Governments can also incentivize sustainability by integrating Green AI criteria into public procurement policies. Many AI companies provide cloud computing, software services, and AI-driven analytics to government agencies. By mandating that vendors meet sustainability benchmarks, including operating on carbon-neutral data centers and using energy-efficient AI models, governments can use their purchasing power to set industry-wide standards. Similar policies have already been applied to green building initiatives, where governments require contractors to meet environmental certifications. Applying the same approach to AI could accelerate the adoption of sustainable practices.

Another innovative policy tool is the introduction of carbon credits specifically tailored for AI workloads. Under this system, AI companies could offset emissions by investing in renewable energy projects or carbon capture technologies. AI firms exceeding predefined emissions thresholds would be required to purchase carbon credits, creating a market-based mechanism that naturally incentivizes efficiency. This concept aligns with broader cap-and-trade programs that have successfully reduced emissions in industries like manufacturing and energy production. However, as seen with the challenges surrounding unbundled Energy Attribute Certificates (EACs), carbon credit programs must be carefully structured to ensure genuine emissions reductions rather than allowing companies to simply "buy their way out" of sustainability commitments.

While government incentives offer powerful mechanisms for promoting Green AI, their design and implementation require careful consideration. Incentives should be structured to drive meaningful change without creating loopholes that allow organizations to claim benefits without genuine improvements in sustainability. Additionally, policies must remain flexible enough to accommodate rapid advancements in AI technology. By strategically combining tax incentives, funding programs, procurement policies, and carbon credit systems, governments can create an ecosystem where sustainability is not just a regulatory requirement but an economic advantage.

17.8.4 Self-Regulation

While government policies play a crucial role in shaping sustainable AI practices, the AI industry itself has the power to drive significant environmental improvements through self-regulation. Many leading AI companies and research organizations have already adopted voluntary commitments to reduce their carbon footprints, improve energy efficiency, and promote sustainable development. These efforts can complement regulatory policies and, in some cases, even set higher standards than those mandated by governments.

One of the most visible self-regulation strategies is the commitment by major AI companies to operate on renewable energy. Companies like Google, Microsoft, Amazon, and Meta have pledged to procure enough clean energy to match 100% of their electricity consumption. Google has gone further by aiming for 24/7 Carbon-Free Energy by ensuring that its data centers run exclusively on renewables every hour of every day. These commitments not only reduce operational emissions but also create market demand for renewable energy, accelerating the transition to a greener grid. However, as seen with the use of unbundled EACs, transparency and accountability in renewable energy claims remain critical to ensuring genuine decarbonization rather than superficial offsets.

Another form of self-regulation is the internal adoption of carbon pricing models. Some companies implement shadow pricing, where they assign an internal cost to carbon emissions in financial decision-making. By incorporating these costs into budgeting and investment strategies, AI companies can prioritize energy-efficient infrastructure and low-emission AI models. This approach mirrors broader corporate sustainability efforts in industries like aviation and manufacturing, where internal carbon pricing has proven to be an effective tool for driving emissions reductions.

Beyond energy consumption, AI developers can implement voluntary efficiency checklists that guide sustainable design choices. Organizations like the AI Sustainability Coalition have proposed frameworks that outline best practices for model development, hardware selection, and operational energy management. These checklists can serve as practical tools for AI engineers to integrate sustainability into their workflows. Companies that publicly commit to following these guidelines set an example for the broader industry, demonstrating that sustainability is not just an afterthought but a core design principle.

Independent sustainability audits further enhance accountability by providing third-party evaluations of AI companies' environmental impact. Firms specializing in technology sustainability, such as Carbon Trust and Green Software Foundation, offer audits that assess energy consumption, carbon emissions, and adherence to green computing best practices. AI companies that voluntarily undergo these audits and publish their findings help build trust with consumers, investors, and regulators. Transparency in environmental reporting allows stakeholders to verify whether companies are meeting their sustainability commitments.

Self-regulation in AI sustainability also extends to open-source collaborations. Initiatives like CodeCarbon and ML CO_2 Impact provide tools that allow developers to estimate and track the carbon footprint of their AI models. By integrating these tools into mainstream AI development platforms like TensorFlow and PyTorch, the industry can normalize sustainability tracking as a standard practice. Encouraging developers to measure and optimize their energy consumption fosters a culture of accountability and continuous improvement.

While self-regulation is an important step toward sustainability, it cannot replace government oversight. Voluntary commitments are only as strong as the incentives driving them, and without external accountability, some companies may prioritize profit over sustainability. However, when combined with regulatory frameworks, self-regulation can accelerate progress by allowing industry leaders to set higher standards than those mandated by law. By embedding sustainability into corporate strategy, AI companies can demonstrate that technological advancement and environmental responsibility are not mutually exclusive.

17.8.5 Global Impact

While AI sustainability efforts are gaining traction, they remain fragmented across national policies, industry initiatives, and regional energy infrastructures. AI's environmental footprint is inherently global, spanning supply chains, cloud data centers, and international markets. A lack of coordination between governments and corporations risks inefficiencies, contradictory regulations, and loopholes that allow companies to shift environmental burdens rather than genuinely reduce them. Establishing global frameworks for AI sustainability is therefore crucial for aligning policies, ensuring accountability, and fostering meaningful progress in mitigating AI's environmental impact.

One of the primary challenges in global AI sustainability efforts is regulatory divergence. Countries and regions are taking vastly different approaches to AI governance. The European Union's AI Act, for example, introduces comprehensive risk-based regulations that include provisions for energy efficiency and environmental impact assessments for AI systems. By contrast, the United States has largely adopted a market-driven approach, emphasizing corporate self-regulation and voluntary sustainability commitments rather than enforceable mandates. Meanwhile, China has prioritized AI dominance through heavy government investment, with sustainability playing a secondary role to technological leadership. This regulatory patchwork creates inconsistencies in how AI-related emissions, resource consumption, and energy efficiency are tracked and managed.

One proposed solution to this fragmentation is the standardization of sustainability reporting metrics for AI systems. Organizations such as the OECD, IEEE, and United Nations have pushed for unified environmental impact reporting standards similar to financial disclosure frameworks. This would allow companies to track and compare their carbon footprints, energy usage, and resource consumption using common methodologies. The adoption of LCA standards for AI, as observed in wider environmental accounting practices, would enable more accurate assessments of AI's total environmental impact, from hardware manufacturing to deployment and decommissioning.

Beyond reporting, energy grid decarbonization remains a critical global consideration. The sustainability of AI is heavily influenced by the carbon intensity of electricity in different regions. For example, training a large AI model in a coal-powered region like Poland results in significantly higher carbon emissions than training the same model in hydroelectric-powered Norway. However, market-based energy accounting practices, including the purchase of unbundled Energy Attribute Certificates (EACs), have allowed some companies to claim carbon neutrality despite operating in high-emission grids. This has led to concerns that sustainability claims may not always reflect actual emissions reductions but instead rely on financial instruments that shift carbon responsibility rather than eliminating it. As a response, Google has championed 24/7 Carbon-Free Energy (CFE), which aims to match local energy consumption with renewable sources in real-time rather than relying on distant offsets. If widely adopted, this model could become a global benchmark for AI sustainability accounting.

Another key area of global concern is AI hardware supply chains and electronic waste management. The production of AI accelerators, GPUs, and data center hardware depends on a complex network of raw material extraction, semiconductor fabrication, and electronic assembly spanning multiple continents. The environmental impact of this supply chain, which includes rare-earth mineral mining in Africa, chip manufacturing in Taiwan, and final assembly in China, often falls outside the jurisdiction of AI companies themselves. This underscores the need for international agreements on sustainable semiconductor production, responsible mining practices, and e-waste recycling policies.

The Basel Convention[14], which regulates hazardous waste exports, could provide a model for addressing AI-related e-waste challenges at a global scale. The convention restricts the transfer of toxic electronic waste from developed nations to developing countries, where unsafe recycling practices can harm workers and pollute local ecosystems. Expanding such agreements to cover AI-specific hardware components, such as GPUs and inference chips, could ensure that end-of-life disposal is handled responsibly rather than outsourced to regions with weaker environmental protections.

International collaboration in AI sustainability is not just about mitigating harm but also leveraging AI as a tool for environmental progress. AI models are already being deployed for climate forecasting, renewable energy optimization, and precision agriculture, demonstrating their potential to contribute to global sustainability goals. Governments, research institutions, and industry leaders must align on best practices for scaling AI solutions that support climate action, ensuring that AI is not merely a sustainability challenge but also a powerful tool for global environmental resilience.

Ultimately, sustainable AI requires a coordinated global approach that integrates regulatory alignment, standardized sustainability reporting, energy decarbonization, supply chain accountability, and responsible e-waste management. Without such collaboration, regional disparities in AI governance could hinder meaningful progress, allowing inefficiencies and externalized environmental costs to persist. As AI continues to evolve, establishing global frameworks that balance technological advancement with environmental responsibility will be critical in shaping an AI-driven future that is not only intelligent but also sustainable.

17.9 Public Engagement

As artificial intelligence (AI) becomes increasingly intertwined with efforts to address environmental challenges, public perception plays a pivotal role in shaping its adoption, regulation, and long-term societal impact. While AI is often viewed as a powerful tool for advancing sustainability, through applications including smart energy management, climate modeling, and conservation efforts, it also faces scrutiny over its environmental footprint, ethical concerns, and transparency.

Public discourse surrounding AI and sustainability is often polarized. On one side, AI is heralded as a transformative force capable of accelerating climate action, reducing carbon emissions, and optimizing resource use. On the other, concerns persist about the high energy consumption of AI models,

[14] Basel Convention: An international treaty regulating the transboundary movement of hazardous waste to prevent its disposal in countries with weaker environmental protections.

the potential for unintended environmental consequences, and the opaque nature of AI-driven decision-making. These contrasting viewpoints influence policy development, funding priorities, and societal acceptance of AI-driven sustainability initiatives.

Bridging the gap between AI researchers, policymakers, and the public is essential for ensuring that AI's contributions to sustainability are both scientifically grounded and socially responsible. This requires clear communication about AI's capabilities and limitations, greater transparency in AI decision-making processes, and mechanisms for inclusive public participation. Without informed public engagement, misunderstandings and skepticism could hinder the adoption of AI solutions that have the potential to drive meaningful environmental progress.

17.9.1 AI Awareness

Public understanding of AI and its role in sustainability remains limited, often shaped by media narratives that highlight either its transformative potential or its risks. Surveys such as the Pew Research Center poll found that while a majority of people have heard of AI, their understanding of its specific applications, especially in the context of sustainability, remains shallow. Many associate AI with automation, recommendation systems, or chatbots but may not be aware of its broader implications in climate science, energy optimization, and environmental monitoring.

A key factor influencing public perception is the framing of AI's sustainability contributions. Optimistic portrayals emphasize AI's ability to enhance renewable energy integration, improve climate modeling accuracy, and enable smart infrastructure for reduced emissions. Organizations such as Climate Change AI actively promote AI's potential in environmental applications, fostering a positive narrative. Conversely, concerns about AI's energy-intensive training processes, ethical considerations, and potential biases contribute to skepticism. Studies analyzing public discourse on AI sustainability reveal an even split between optimism and caution, with some fearing that AI's environmental costs may outweigh its benefits.

In many cases, public attitudes toward AI-driven sustainability efforts are shaped by trust in institutions. AI systems deployed by reputable environmental organizations or in collaboration with scientific communities tend to receive more favorable reception. However, corporate-led AI sustainability initiatives often face skepticism, particularly if they are perceived as greenwashing—a practice where companies exaggerate their commitment to environmental responsibility without substantial action.

To foster informed public engagement, increasing AI literacy is crucial. This involves education on AI's actual energy consumption, potential for optimization, and real-world applications in sustainability. Universities, research institutions, and industry leaders can play a pivotal role in making AI's sustainability impact more accessible to the general public through open reports, interactive tools, and clear communication strategies.

17.9.2 Messaging and Discourse

How AI is communicated to the public significantly influences perceptions of its role in sustainability. The messaging around AI-driven environmental efforts must balance technical accuracy, realistic expectations, and transparency to ensure constructive discourse.

Optimistic narratives emphasize AI's potential as a powerful tool for sustainability. Initiatives such as Climate Change AI and AI-driven conservation projects highlight applications in wildlife protection, climate modeling, energy efficiency, and pollution monitoring. These examples are often framed as AI augmenting human capabilities, enabling more precise and scalable solutions to environmental challenges. Such positive framing encourages public support and investment in AI-driven sustainability research.

However, skepticism remains, particularly regarding AI's own environmental footprint. Critical perspectives highlight the massive energy demands of AI model training, particularly for large-scale neural networks. The Asilomar AI Principles and other cautionary frameworks stress the need for transparency, ethical guardrails, and energy-conscious AI development. The rise of generative AI models has further amplified concerns about data center energy consumption, supply chain sustainability, and the long-term viability of compute-intensive AI workloads.

A key challenge in AI sustainability messaging is avoiding extremes. Public discourse often falls into two polarized views: one where AI is seen as an indispensable tool for solving climate change, and another where AI is portrayed as an unchecked technology accelerating ecological harm. Neither view fully captures the nuanced reality. AI, like any technology, is a tool whose environmental impact depends on how it is developed, deployed, and governed.

To build public trust and engagement, AI sustainability messaging should prioritize three key aspects. First, it must acknowledge clear trade-offs by presenting both the benefits and limitations of AI for sustainability, including energy consumption, data biases, and real-world deployment challenges. Second, messaging should rely on evidence-based claims, communicating AI's impact through data-driven assessments, lifecycle analyses, and transparent carbon accounting rather than speculative promises. Third, the framing should remain human-centered, emphasizing collaborative AI systems that work alongside scientists, policymakers and communities rather than fully automated, opaque decision-making systems. Through this balanced, transparent approach, AI can maintain credibility while driving meaningful environmental progress.

Effective public engagement relies on bridging the knowledge gap between AI practitioners and non-experts, ensuring that AI's role in sustainability is grounded in reality, openly discussed, and continuously evaluated.

17.9.3 Transparency and Trust

As AI systems become more integrated into sustainability efforts, transparency and trust are crucial for ensuring public confidence in their deployment. The complexity of AI models, particularly those used in environmental monitoring, resource optimization, and emissions tracking, often makes it difficult for stakeholders to understand how decisions are being made. Without clear explanations of how AI systems operate, concerns about bias, accountability, and unintended consequences can undermine public trust.

A key aspect of transparency involves ensuring that AI models used in sustainability applications are explainable and interpretable. The National Institute of Standards and Technology (NIST) Principles for Explainable AI provide a framework for designing systems that offer meaningful and understandable explanations of their outputs. These principles emphasize that AI-generated decisions should be contextually relevant, accurately reflect the model's logic, and clearly communicate the limitations of the system (Phillips et al. 2020). In sustainability applications, where AI influences environmental policy, conservation strategies, and energy management, interpretability is essential for public accountability.

Transparency is also necessary in AI sustainability claims. Many technology companies promote AI-driven sustainability initiatives, yet without standardized reporting, it is difficult to verify the actual impact. The Montréal Carbon Pledge offers a valuable framework for accountability in this space:

> "As institutional investors, we must act in the best long-term interests of our beneficiaries. In this fiduciary role, long-term investment risks are associated with greenhouse gas emissions, climate change, and carbon regulation. Measuring our carbon footprint is integral to understanding better, quantifying, and managing the carbon and climate change-related impacts, risks, and opportunities in our investments. Therefore, as a first step, we commit to measuring and disclosing the carbon footprint of our investments annually to use this information to develop an engagement strategy and identify and set carbon footprint reduction targets." — Montréal Carbon Pledge

This commitment to measuring and disclosing carbon footprints serves as a model for how AI sustainability claims could be validated. A similar commitment for AI, where companies disclose the environmental footprint of training and deploying models, would provide the public with a clearer picture of AI's sustainability contributions. Without such measures, companies risk accusations of "greenwashing," where claims of sustainability benefits are exaggerated or misleading.

Beyond corporate accountability, transparency in AI governance ensures that AI systems deployed for sustainability are subject to ethical oversight. The integration of AI into environmental decision-making raises questions about who has control over these technologies and how they align with societal values. Efforts such as the OECD AI Policy Observatory highlight the need for regulatory frameworks that require AI developers to disclose energy consumption, data sources, and model biases when deploying AI in critical sustainability applications. Public accessibility to this information would enable greater scrutiny and foster trust in AI-driven solutions.

Building trust in AI for sustainability requires not only clear explanations of how models function but also proactive efforts to include stakeholders in decision-making processes. Transparency mechanisms such as open-access datasets, public AI audits, and participatory model development can enhance accountability. By ensuring that AI applications in sustainability remain understandable, verifiable, and ethically governed, trust can be established, enabling broader public support for AI-driven environmental solutions.

17.9.4 Engagement and Awareness

Public engagement plays a crucial role in shaping the adoption and effectiveness of AI-driven sustainability efforts. While AI has the potential to drive significant environmental benefits, its success depends on how well the public understands and supports its applications. Widespread misconceptions, limited awareness of AI's role in sustainability, and concerns about ethical and environmental risks can hinder meaningful engagement. Addressing these issues requires deliberate efforts to educate, involve, and empower diverse communities in discussions about AI's impact on environmental sustainability.

Surveys indicate that while AI is widely recognized, the specific ways it intersects with sustainability remain unclear to the general public. A study conducted by the Pew Research Center found that while 87% of respondents had some awareness of AI, only a small fraction could explain how it affects energy consumption, emissions, or conservation efforts. This gap in understanding can lead to skepticism, with some viewing AI as a potential contributor to environmental harm due to its high computational demands rather than as a tool for addressing climate challenges. To build public confidence in AI sustainability initiatives, clear communication is essential.

Efforts to improve AI literacy in sustainability contexts can take multiple forms. Educational campaigns highlighting AI's role in optimizing renewable energy grids, reducing food waste, or monitoring biodiversity can help demystify the technology. Programs such as Climate Change AI and Partnership on AI actively work to bridge this gap by providing accessible research, case studies, and policy recommendations that illustrate AI's benefits in addressing climate change. Similarly, media representation plays a significant role in shaping perceptions, and responsible reporting on AI's environmental potential, in conjunction with its challenges, can provide a more balanced narrative.

Beyond education, engagement requires active participation from various stakeholders, including local communities, environmental groups, and policymakers. Many AI-driven sustainability projects focus on data collection and automation but lack mechanisms for involving affected communities in decision-making. For example, AI models used in water conservation or wildfire prediction may rely on data that overlooks the lived experiences of local populations. Creating channels for participatory AI design, in which communities contribute insights, validate model outputs, and influence policy, can lead to more inclusive and context-aware sustainability solutions.

Transparency and public input are particularly important when AI decisions affect resource allocation, environmental justice, or regulatory actions. AI-driven carbon credit markets, for instance, require mechanisms to ensure that communities in developing regions benefit from sustainability initiatives rather than facing unintended harms such as land displacement or exploitation. Public consultations, open-data platforms, and independent AI ethics committees can help integrate societal values into AI-driven sustainability policies.

Ultimately, fostering public engagement and awareness in AI sustainability requires a multi-faceted approach that combines education, communication, and participatory governance. By ensuring that AI systems are accessible, understandable, and responsive to community needs, public trust and support for AI-driven sustainability solutions can be strengthened. This engagement is essential to aligning AI innovation with societal priorities and ensuring that environmental AI systems serve the broader public good.

17.9.5 Equitable AI Access

Ensuring equitable access to AI-driven sustainability solutions is essential for fostering global environmental progress. While AI has demonstrated its ability to optimize energy grids, monitor deforestation, and improve climate modeling, access to these technologies remains unevenly distributed. Developing nations, marginalized communities, and small-scale environmental organizations often lack the infrastructure, funding, and expertise necessary to leverage AI effectively. Addressing these disparities is crucial to ensuring that the benefits of AI sustainability solutions reach all populations rather than exacerbating existing environmental and socio-economic inequalities.

One of the primary barriers to equitable AI access is the digital divide. Many AI sustainability applications rely on advanced computing infrastructure, cloud resources, and high-quality datasets, which are predominantly concentrated in high-income regions. A recent OECD report on national AI compute capacity highlighted that many countries lack a strategic roadmap for developing AI infrastructure, leading to a growing gap between AI-rich and AI-poor regions (Oecd 2023). Without targeted investment in AI infrastructure, lower-income countries remain excluded from AI-driven sustainability advancements. Expanding access to computing resources, supporting open-source AI frameworks, and providing cloud-based AI solutions for environmental monitoring could help bridge this gap.

In addition to infrastructure limitations, a lack of high-quality, region-specific data poses a significant challenge. AI models trained on datasets from industrialized nations may not generalize well to other geographic and socio-economic contexts. For example, an AI model optimized for water conservation in North America may be ineffective in regions facing different climate patterns, agricultural practices, or regulatory structures. Efforts to localize AI sustainability applications, through the collection of diverse datasets, partnerships with local organizations, and the integration of indigenous knowledge, can enhance the relevance and impact of AI solutions in underrepresented regions.

Access to AI tools also requires technical literacy and capacity-building initiatives. Many small environmental organizations and community-driven sustainability projects do not have the in-house expertise needed to develop or deploy AI solutions effectively. Capacity-building efforts, such as AI training programs, knowledge-sharing networks, and collaborations between academic institutions and environmental groups, can empower local stakeholders to adopt AI-driven sustainability practices. Organizations like Climate Change AI and the Partnership on AI have taken steps to provide resources and guidance on using AI for environmental applications, but more widespread efforts are needed to democratize access.

Funding mechanisms also play a critical role in determining who benefits from AI-driven sustainability. While large corporations and well-funded research institutions can afford to invest in AI-powered environmental solutions, smaller organizations often lack the necessary financial resources. Government grants, philanthropic funding, and international AI-for-good initiatives could help ensure that grassroots sustainability efforts can leverage AI technologies. For instance, Spain has allocated 300 million euros specifically for AI and sustainability projects, setting a precedent for public investment in environmentally responsible AI innovation. Expanding such funding models globally could foster more inclusive AI adoption.

Beyond technical and financial barriers, policy interventions are necessary to ensure that AI sustainability efforts are equitably distributed. Without regulatory frameworks that prioritize inclusion, AI-driven environmental solutions may disproportionately benefit regions with existing technological advantages while neglecting areas with the most pressing sustainability challenges. Governments and international bodies should establish policies that encourage equitable AI adoption, such as requiring AI sustainability projects to consider social impact assessments or mandating transparent reporting on AI-driven environmental initiatives.

Ensuring equitable access to AI for sustainability is not merely a technical challenge but a fundamental issue of environmental justice. As AI continues to shape global sustainability efforts, proactive measures must be taken to prevent technology from reinforcing existing inequalities. By investing in AI infrastructure, localizing AI applications, supporting capacity-building efforts, and implementing inclusive policies, AI can become a tool that empowers all communities in the fight against climate change and environmental degradation.

17.10 Future Challenges

As AI continues to evolve, its role in environmental sustainability is set to expand. Advances in AI have the potential to accelerate progress in renewable energy, climate modeling, biodiversity conservation, and resource efficiency. However, realizing this potential requires addressing significant challenges related to energy efficiency, infrastructure sustainability, data availability, and governance. The future of AI and sustainability hinges on balancing innovation with responsible environmental stewardship, ensuring that AI-driven progress does not come at the cost of increased environmental degradation.

17.10.1 Future Directions

A major priority in AI sustainability is the development of more energy-efficient models and algorithms. Optimizing deep learning models to minimize computational cost is a key research direction, with techniques such as model pruning, quantization, and low-precision numerics demonstrating significant potential for reducing energy consumption without compromising performance. These strategies aim to improve the efficiency of AI workloads while leveraging specialized hardware accelerators to maximize computational throughput with minimal energy expenditure. The continued development of non-von Neumann computing paradigms, such as neuromorphic computing and in-memory computing, presents another avenue for energy-efficient AI architectures, as explored in the Hardware Acceleration chapter.

Another crucial direction involves the integration of renewable energy into AI infrastructure. Given that data centers are among the largest contributors to AI's carbon footprint, shifting towards clean energy sources like solar, wind, and hydroelectric power is imperative. The feasibility of this transition depends on advancements in sustainable energy storage technologies, such as those being developed by companies like Ambri, an MIT spinoff working on liquid metal battery solutions. These innovations could enable data centers to operate on renewable energy with greater reliability, reducing dependency on fossil fuel-based grid power. However, achieving this transition at scale requires collaborative efforts between AI companies, energy providers, and policymakers to develop grid-aware AI scheduling and carbon-aware workload management strategies, ensuring that compute-intensive AI tasks are performed when renewable energy availability is at its peak.

Beyond energy efficiency, AI sustainability will also benefit from intelligent resource allocation and waste reduction strategies. Improving the utilization of computing resources, reducing redundant model training cycles, and implementing efficient data sampling techniques can substantially decrease energy consumption. A key challenge in AI model development is the trade-off between experimentation and efficiency—techniques such as neural architecture search and hyperparameter optimization can improve model performance but often require vast computational resources. Research into efficient experimentation methodologies could help strike a balance, allowing for model improvements while mitigating the environmental impact of excessive training runs.

17.10.2 Challenges

Despite these promising directions, significant obstacles must be addressed to make AI truly sustainable. One of the most pressing challenges is the lack of standardized measurement and reporting frameworks for evaluating AI's environmental footprint. Unlike traditional industries, where LCA methodologies are well-established, AI systems require more comprehensive and adaptable approaches that account for the full environmental impact of both hardware (compute infrastructure) and software (model training and inference cycles). While efforts such as MLCommons have begun integrating energy efficiency into benchmarking practices, a broader, globally recognized standard is necessary to ensure consistency in reporting AI-related emissions.

Another critical challenge is optimizing AI infrastructure for longevity and sustainability. AI accelerators and data center hardware must be designed with maximized utilization, extended operational lifespans, and minimal environmental impact in mind. Unlike conventional hardware refresh cycles, which often prioritize performance gains over sustainability, future AI infrastructure must prioritize reusability, modular design, and circular economy principles to minimize electronic waste and reduce reliance on rare earth materials.

From a software perspective, minimizing redundant computation is essential to reducing energy-intensive workloads. The practice of training larger models on increasingly vast datasets, while beneficial for accuracy, comes with diminishing returns in sustainability. A data-centric approach to AI model development, as highlighted in recent work (C.-J. Wu et al. 2022), suggests that the predictive value of data decays over time, making it crucial to identify and filter the most relevant data subsets. Smarter data sampling strategies can optimize training processes, ensuring that only the most informative data is used to refine models, reducing the energy footprint without sacrificing model quality.

A further challenge lies in data accessibility and transparency. Many AI sustainability efforts rely on corporate and governmental disclosures of energy usage, carbon emissions, and environmental impact data. However, data gaps and inconsistencies hinder efforts to accurately assess AI's footprint. Greater transparency from AI companies regarding their sustainability initiatives, coupled with open-access datasets for environmental impact research, would enable more rigorous analysis and inform best practices for sustainable AI development.

Finally, the rapid pace of AI innovation poses challenges for regulation and governance. Policymakers must develop agile, forward-looking policies that promote sustainability while preserving the flexibility needed for AI research and innovation. Regulatory frameworks should encourage efficient AI practices, such as promoting carbon-aware computing, incentivizing energy-efficient AI model development, and ensuring that AI-driven environmental applications align with broader sustainability goals. Achieving this requires close collaboration between AI researchers, environmental scientists, energy sector stakeholders, and policymakers to develop a regulatory landscape that fosters responsible AI growth while minimizing ecological harm.

17.10.3 Towards Sustainable AI

The future of AI in sustainability is both promising and fraught with challenges. To harness AI's full potential while mitigating its environmental impact, the field must embrace energy-efficient model development, renewable energy integration, hardware and software optimizations, and transparent environmental reporting. Addressing these challenges will require multidisciplinary collaboration across technical, industrial, and policy domains, ensuring that AI's trajectory aligns with global sustainability efforts.

By embedding sustainability principles into AI system design, optimizing compute infrastructure, and establishing clear accountability mechanisms, AI can serve as a catalyst for environmental progress rather than a contributor to ecological degradation. The coming years will be pivotal in shaping AI's role in sustainability, determining whether it amplifies existing challenges or emerges as a key tool in the fight against climate change and resource depletion.

17.11 Conclusion

The integration of AI into environmental sustainability presents both immense opportunities and formidable challenges. As AI systems continue to scale in complexity and influence, their environmental footprint must be addressed through energy-efficient design, responsible infrastructure deployment, transparent accountability measures, and policy-driven interventions. While AI offers powerful capabilities for climate modeling, emissions reduction, resource optimization, and biodiversity conservation, its reliance on compute-intensive hardware, large-scale data processing, and energy-hungry model training necessitates a careful balance between progress and sustainability.

This chapter has explored the full lifecycle impact of AI systems, from their carbon footprint and energy consumption to hardware manufacturing, e-waste concerns, and the role of embedded AI in the growing "Internet of Trash." We have examined strategies for mitigating AI's environmental impact, including advances in green AI infrastructure, energy-aware model optimization, and lifecycle-aware AI development. Additionally, we have highlighted the importance of policy and regulatory frameworks in shaping a sustainable AI ecosystem, emphasizing the need for measurement and reporting mandates, incentive structures, and governance mechanisms that align AI innovation with long-term environmental goals.

Public perception and engagement remain central to the discourse on AI and sustainability. Transparent AI practices, explainable models, and ethical governance frameworks will be key to fostering trust and ensuring that AI solutions are inclusive, equitable, and accountable. The responsible deployment of AI in sustainability efforts must incorporate stakeholder input, interdisciplinary collaboration, and a commitment to minimizing unintended consequences.

Looking ahead, the path toward sustainable AI requires continuous advancements in hardware efficiency, carbon-aware computing, renewable energy integration, and equitable access to AI resources. Overcoming challenges such as data gaps, inconsistent environmental reporting, and planned obsolescence in AI hardware will require collective efforts from AI researchers, environmental scientists, policymakers, and industry leaders. By embedding sustainability at the core of AI development, we can ensure that AI not only accelerates technological progress but also contributes meaningfully to a more sustainable and resilient future.

AI has the potential to be a force for good in the fight against climate change and resource depletion, but its long-term impact depends on the choices we make today. Through innovation, regulation, and collective responsibility, AI can evolve as a technology that enhances environmental sustainability rather than exacerbating ecological strain. The decisions made by AI practitioners, policymakers, and society at large will shape whether AI serves as a tool for sustainable progress or an unchecked driver of environmental harm. The imperative now is to act deliberately, designing AI systems that align with global sustainability goals and contribute to a future where technological advancement and ecological well-being coexist harmoniously.

17.12 Resources

ℹ Slides

- *Coming soon.*

❗ Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 18

Robust AI

Figure 18.1: *DALL·E 3 Prompt: Create an image featuring an advanced AI system symbolized by an intricate, glowing neural network, deeply nested within a series of progressively larger and more fortified shields. Each shield layer represents a layer of defense, showcasing the system's robustness against external threats and internal errors. The neural network, at the heart of this fortress of shields, radiates with connections that signify the AI's capacity for learning and adaptation. This visual metaphor emphasizes not only the technological sophistication of the AI but also its resilience and security, set against the backdrop of a state-of-the-art, secure server room filled with the latest in technological advancements. The image aims to convey the concept of ultimate protection and resilience in the field of artificial intelligence.*

Purpose

How do we develop fault-tolerant and resilient machine learning systems for real-world deployment?

The integration of machine learning systems into real-world applications demands fault-tolerant execution. However, these systems are inherently vulnerable to a spectrum of challenges that can degrade their capabilities. From subtle hardware anomalies to sophisticated adversarial attacks and the unpredictable nature of real-world data, the potential for failure is ever-present. This reality underscores the need to fundamentally rethink how AI systems are designed and deployed, placing robustness and trustworthiness at the forefront. Building resilient machine learning systems is not merely a technical objective; it is a foundational requirement for ensuring their safe and effective operation in dynamic and uncertain environments.

Learning Objectives

- Identify common hardware faults impacting AI performance.
- Explain how hardware faults (transient and permanent) affect AI systems.
- Define adversarial attacks and their impact on ML models.
- Recognize the vulnerabilities of ML models to data poisoning.
- Explain the challenges posed by distribution shifts in ML models.
- Describe the role of software fault detection and mitigation in AI systems.
- Understand the importance of a holistic software development approach for robust AI.

18.1 Overview

As ML systems become increasingly integrated into various domains, ranging from cloud-based services to edge devices and embedded systems, the impact of hardware and software faults on their performance and reliability grows more pronounced. Looking ahead, as these systems become more complex and are deployed in safety-critical applications, the need for robust and fault-tolerant designs becomes paramount.

ML systems are expected to play critical roles in autonomous vehicles, smart cities, healthcare, and industrial automation. In these domains, the consequences of systemic failures, including hardware and software faults, and malicious inputs such as adversarial attacks and data poisoning, and environmental shifts, can be severe, potentially resulting in loss of life, economic disruption, or environmental harm.

To address these risks, researchers and engineers must develop advanced techniques for fault detection, isolation, and recovery, ensuring the reliable operation of future ML systems.

Definition of Robust AI

Robust Artificial Intelligence (Robust AI) refers to the ability of AI systems to maintain *performance and reliability* in the presence of *internal and external system errors, and malicious inputs and changes to the data or environment*. Robust AI systems are designed to be *fault-tolerant* and *error-resilient*, capable of functioning effectively despite *variations and errors within the operational environment*. Achieving Robust AI involves strategies for *fault detection, mitigation, and recovery*, as well as prioritizing *resilience throughout the AI development lifecycle*.

We focus specifically on categories of faults and errors that can impact the robustness of ML systems: errors arising from the underlying system, malicious manipulation, and environmental changes.

Systemic hardware failures present significant challenges across computing systems. Whether transient, permanent, or intermittent, these faults can corrupt computations and degrade system performance. The impact ranges from temporary glitches to complete component failures, requiring robust detection and mitigation strategies to maintain reliable operation.

Malicious manipulation of ML models remains a critical concern as ML systems face various threats to their integrity. Adversarial attacks, data poisoning attempts, and distribution shifts can cause models to misclassify inputs, exhibit distorted behavior patterns, or produce unreliable outputs. These vulnerabilities underscore the importance of developing resilient architectures and defensive mechanisms to protect model performance.

Environmental changes introduce another dimension of potential faults that must be carefully managed. Bugs, design flaws, and implementation errors within algorithms, libraries, and frameworks can propagate through the system, creating systemic vulnerabilities. Rigorous testing, monitoring, and quality control processes help identify and address these software-related issues before they impact production systems.

The specific approaches to achieving robustness vary significantly based on deployment context and system constraints. Large-scale cloud computing environments and data centers typically emphasize fault tolerance through redundancy, distributed processing architectures, and sophisticated error detection mechanisms. In contrast, edge devices and embedded systems must address robustness challenges within strict computational, memory, and energy limitations. This necessitates careful optimization and targeted hardening strategies appropriate for resource-constrained environments.

Regardless of deployment context, the essential characteristics of a robust ML system include fault tolerance, error resilience, and sustained performance. By understanding and addressing these multifaceted challenges, it is possible to develop reliable ML systems capable of operating effectively in real-world environments.

This chapter not only explores the tools, frameworks, and techniques used to detect and mitigate faults, attacks, and distribution shifts, but also emphasizes the importance of prioritizing resilience throughout the AI development lifecycle—from data collection and model training to deployment and monitoring. Proactively addressing robustness challenges is key to unlocking the full potential of ML technologies while ensuring their safe, dependable, and responsible deployment.

18.2 Real-World Applications

Understanding the importance of robustness in machine learning systems requires examining how faults manifest in practice. Real-world case studies illustrate the consequences of hardware and software faults across cloud, edge, and embedded environments. These examples highlight the critical need for fault-tolerant design, rigorous testing, and robust system architectures to ensure reliable operation in diverse deployment scenarios.

18.2.1 Cloud

In February 2017, Amazon Web Services (AWS) experienced a significant outage due to human error during routine maintenance. An engineer inadvertently entered an incorrect command, resulting in the shutdown of multiple servers. This outage disrupted many AWS services, including Amazon's AI-powered assistant, Alexa. As a consequence, Alexa-enabled devices, including Amazon Echo and third-party products that utilize Alexa Voice Service, were unresponsive for several hours. This incident underscores the impact of human error on cloud-based ML systems and the importance of robust maintenance protocols and failsafe mechanisms.

In another case (Vangal et al. 2021), Facebook encountered a silent data corruption (SDC) issue in its distributed querying infrastructure, illustrated in Figure 18.2. SDC refers to undetected errors during computation or data transfer that propagate silently through system layers. Facebook's system processed SQL-like queries across datasets and supported a compression application designed to reduce data storage footprints. Files were compressed when not in use and decompressed upon read requests. A size check was performed before decompression to ensure the file was valid. However, an unexpected fault occasionally returned a file size of zero for valid files, leading to decompression failures and missing entries in the output database. The issue appeared sporadically, with some computations returning correct file sizes, making it particularly difficult to diagnose.

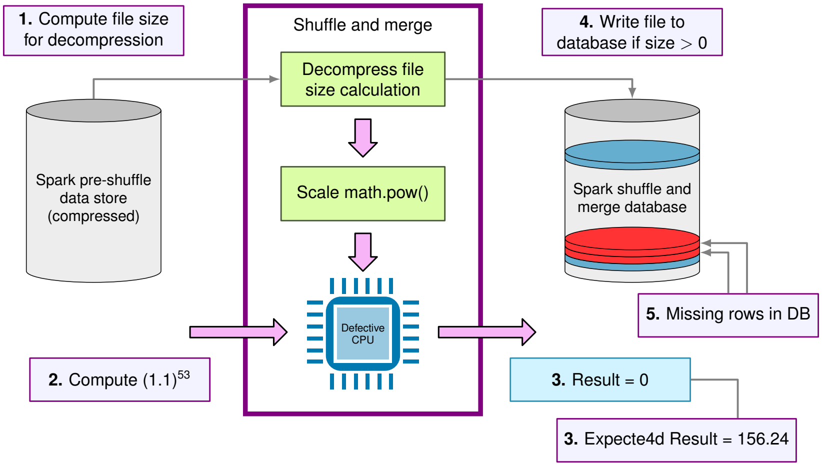
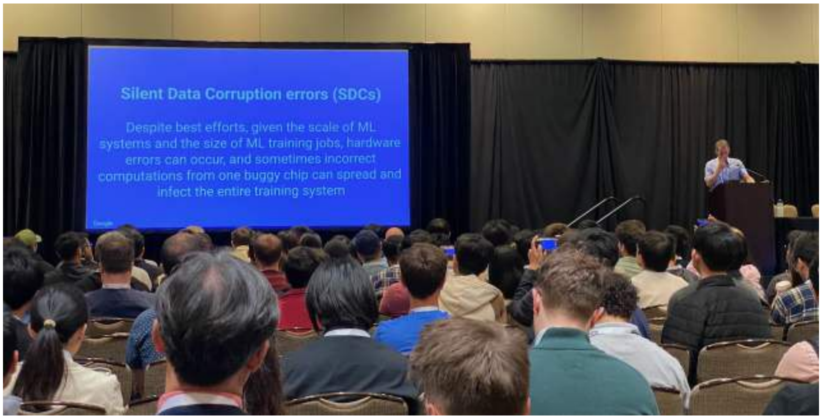

1. Compute file size for decompression
Shuffle and merge
4. Write file to database if size > 0
Decompress file size calculation
Spark pre-shuffle data store (compressed)
Scale math.pow()
Spark shuffle and merge database
Defective CPU
5. Missing rows in DB
2. Compute (1.1)^53
3. Result = 0
3. Expecte4d Result = 156.24

Figure 18.2: Silent data corruption in database applications. Source: Facebook.

This case illustrates how silent data corruption can propagate across multiple layers of the application stack, resulting in data loss and application failures in large-scale distributed systems. Left unaddressed, such errors can degrade ML system performance. For example, corrupted training data or inconsistencies in data pipelines due to SDC may compromise model accuracy and reliability. Similar challenges have been reported by other major companies. As shown in Figure 18.3, Jeff Dean, Chief Scientist at Google DeepMind and Google Research, highlighted these issues in AI hypercomputers during a keynote at MLSys 2024.



Silent Data Corruption errors (SDCs)

Despite best efforts, given the scale of ML systems and the size of ML training jobs, hardware errors can occur, and sometimes incorrect computations from one buggy chip can spread and infect the entire training system

Figure 18.3: Silent data corruption (SDC) errors are a major issue for AI hypercomputers. Source: Jeff Dean at MLSys 2024, Keynote (Google).

18.2.2 Edge

In the edge computing domain, self-driving vehicles provide prominent examples of how faults can critically affect ML systems. These vehicles depend on machine learning for perception, decision-making, and control, making them particularly vulnerable to both hardware and software faults.

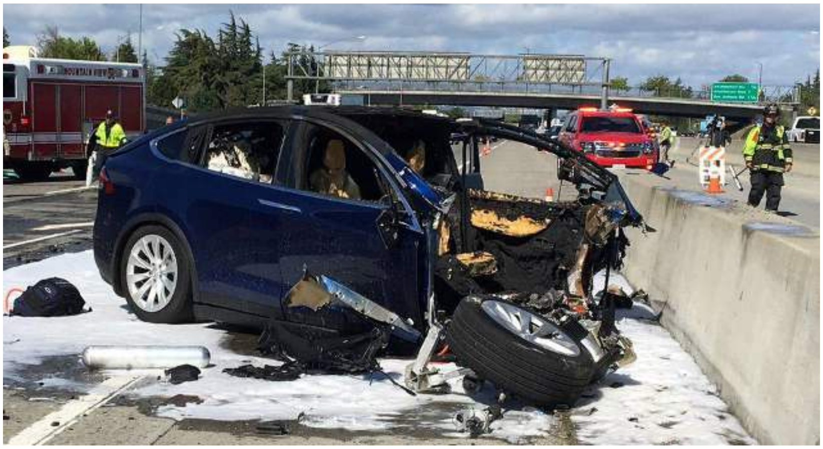


Figure 18.4: Tesla in the fatal California crash was on Autopilot. Source: BBC News

In May 2016, a fatal crash occurred when a Tesla Model S operating in Autopilot mode[0] collided with a white semi-trailer truck. The system, relying on computer vision and ML algorithms, failed to distinguish the trailer against a bright sky, leading to a high-speed impact. The driver, reportedly distracted at the time, did not intervene, as shown in Figure 18.4. This incident raised serious concerns about the reliability of AI-based perception systems and emphasized the need for robust failsafe mechanisms in autonomous vehicles. A similar case occurred in March 2018, when an Uber self-driving test vehicle struck and killed a pedestrian in Tempe, Arizona. The accident was attributed to a flaw in

[0] Autopilot: Tesla's driver assistance system that provides semi-autonomous capabilities like steering, braking, and acceleration while requiring active driver supervision.

the vehicle's object recognition software, which failed to classify the pedestrian as an obstacle requiring avoidance.

18.2.3 Embedded

Embedded systems operate in resource-constrained and often safety-critical environments. As AI capabilities are increasingly integrated into these systems, the complexity and consequences of faults grow significantly.

One example comes from space exploration. In 1999, NASA's Mars Polar Lander mission experienced a catastrophic failure due to a software error in its touchdown detection system (Figure 18.5). The lander's software misinterpreted the vibrations from the deployment of its landing legs as a successful touchdown, prematurely shutting off its engines and causing a crash. This incident underscores the importance of rigorous software validation and robust system design, particularly for remote missions where recovery is impossible. As AI becomes more integral to space systems, ensuring robustness and reliability will be essential to mission success.

Figure 18.5: NASA's Failed Mars Polar Lander mission in 1999 cost over $200M. Source: SlashGear

Another example occurred in 2015, when a Boeing 787 Dreamliner experienced a complete electrical shutdown mid-flight due to a software bug in its generator control units. The failure stemmed from a scenario in which powering up all four generator control units simultaneously, following 248 days of uninterrupted operation, caused them to enter failsafe mode, disabling all AC electrical power.

> *"If the four main generator control units (associated with the engine-mounted generators) were powered up at the same time, after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase." — Federal Aviation Administration directive (2015)*

As AI is increasingly applied in aviation, including tasks such as autonomous flight control and predictive maintenance, the robustness of embedded systems becomes critical for passenger safety.

Finally, consider the case of implantable medical devices. For instance, a smart pacemaker that experiences a fault or unexpected behavior due to software or hardware failure could place a patient's life at risk. As AI systems take on perception, decision-making, and control roles in such applications, new sources of vulnerability emerge, including data-related errors, model uncertainty[1], and unpredictable behaviors in rare edge cases. Moreover, the opaque nature of some AI models complicates fault diagnosis and recovery.

18.3 Hardware Faults

Hardware faults are a significant challenge in computing systems, including both traditional and ML systems. These faults occur when physical components, including processors, memory modules, storage devices, and interconnects, malfunction or behave abnormally. Hardware faults can cause incorrect computations, data corruption, system crashes, or complete system failure, compromising the integrity and trustworthiness of the computations performed by the system (S. Jha et al. 2019). A complete system failure refers to a situation where the entire computing system becomes unresponsive or inoperable due to a critical hardware malfunction. This type of failure is the most severe, as it renders the system unusable and may lead to data loss or corruption, requiring manual intervention to repair or replace the faulty components.

ML systems depend on complex hardware architectures and large-scale computations to train and deploy models that learn from data and make intelligent predictions. As a result, hardware faults can disrupt the MLOps pipeline, introducing errors that compromise model accuracy, robustness, and reliability (G. Li et al. 2017). Understanding the types of hardware faults, their mechanisms, and their impact on system behavior is essential for developing strategies to detect, mitigate, and recover from these issues.

The following sections will explore the three main categories of hardware faults: transient, permanent, and intermittent. We will discuss their definitions, characteristics, causes, mechanisms, and examples of how they manifest in computing systems. Detection and mitigation techniques specific to each fault type will also be covered.

- **Transient Faults**: Transient faults are temporary and non-recurring. They are often caused by external factors such as cosmic rays, electromagnetic interference, or power fluctuations. A common example of a transient fault is a bit flip, where a single bit in a memory location or register changes its value unexpectedly. Transient faults can lead to incorrect computations or data corruption, but they do not cause permanent damage to the hardware.
- **Permanent Faults**: Permanent faults, also called hard errors, are irreversible and persist over time. They are typically caused by physical defects or wear-out of hardware components. Examples of permanent faults include stuck-at faults, where a bit or signal is permanently set to

[1] Model Uncertainty: The inadequacy of a machine learning model to capture the full complexity of the underlying data-generating process.

a specific value (e.g., always 0 or always 1), and device failures, such as a malfunctioning processor or a damaged memory module. Permanent faults can result in complete system failure or significant performance degradation.

- **Intermittent Faults**: Intermittent faults are recurring faults that appear and disappear intermittently. Unstable hardware conditions, such as loose connections, aging components, or manufacturing defects, often cause them. Intermittent faults can be challenging to diagnose and reproduce because they may occur sporadically and under specific conditions. Examples include intermittent short circuits or contact resistance issues. These faults can lead to unpredictable system behavior and sporadic errors.

Understanding this fault taxonomy and its relevance to both traditional computing and ML systems provides a foundation for making informed decisions when designing, implementing, and deploying fault-tolerant solutions. This knowledge is crucial for improving the reliability and trustworthiness of computing systems and ML applications.

18.3.1 Transient Faults

Transient faults in hardware can manifest in various forms, each with its own unique characteristics and causes. These faults are temporary in nature and do not result in permanent damage to the hardware components.

18.3.1.1 Characteristics

All transient faults are characterized by their short duration and non-permanent nature. They do not persist or leave any lasting impact on the hardware. However, they can still lead to incorrect computations, data corruption, or system misbehavior if not properly handled. A classic example is shown in Figure 18.6, where a single bit in memory unexpectedly changes state, potentially altering critical data or computations.

Some of the common types of transient faults include Single Event Upsets (SEUs) caused by ionizing radiation, voltage fluctuations (Reddi and Gupta 2013) due to power supply noise or electromagnetic interference, Electromagnetic Interference (EMI) induced by external electromagnetic fields, Electrostatic Discharge (ESD) resulting from sudden static electricity flow, crosstalk caused by unintended signal coupling, ground bounce triggered by simultaneous switching of multiple outputs, timing violations due to signal timing constraint breaches, and soft errors in combinational logic affecting the output of logic circuits (Mukherjee, Emer, and Reinhardt, n.d.). Understanding these different types of transient faults is crucial for designing robust and resilient hardware systems that can mitigate their impact and ensure reliable operation.

18.3.1.2 Causes

Transient faults can be attributed to various external factors. One common cause is cosmic rays—high-energy particles originating from outer space. When these particles strike sensitive areas of the hardware, such as memory cells or transistors, they can induce charge disturbances that alter the stored or

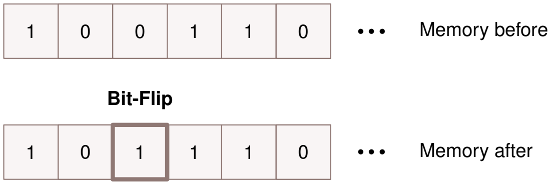


Figure 18.6: An illustration of a bit-flip error, where a single bit in memory changes state, leading to data corruption or computation errors.

transmitted data. This is illustrated in Figure 18.7. Another cause of transient faults is electromagnetic interference (EMI) from nearby devices or power fluctuations. EMI can couple with the circuits and cause voltage spikes or glitches that temporarily disrupt the normal operation of the hardware.

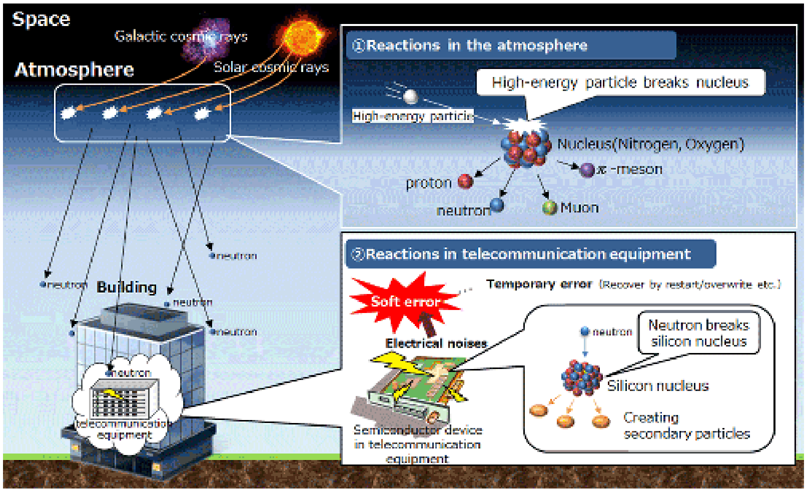

Space
Atmosphere
Galactic cosmic rays
Solar cosmic rays
①Reactions in the atmosphere
High-energy particle breaks nucleus
High-energy particle
Nucleus(Nitrogen, Oxygen)
π -meson
proton
neutron
Muon
neutron
neutron
Building
neutron
neutron
neutron
telecommunication equipment
②Reactions in telecommunication equipment
Soft error
Temporary error (Recover by restart/overwrite etc.)
Electrical noises
neutron
Neutron breaks silicon nucleus
Silicon nucleus
Creating secondary particles
Semiconductor device in telecommunication equipment

Figure 18.7: Mechanism of Hardware Transient Fault Occurrence. Source: NTT

18.3.1.3 Mechanisms

Transient faults can manifest through different mechanisms depending on the affected hardware component. In memory devices like DRAM or SRAM, transient faults often lead to bit flips, where a single bit changes its value from 0 to 1 or vice versa. This can corrupt the stored data or instructions. In logic circuits, transient faults can cause glitches[2] or voltage spikes propagating through the combinational logic[3], resulting in incorrect outputs or control signals. Transient faults can also affect communication channels, causing bit errors or packet losses during data transmission.

18.3.1.4 Impact on ML

A common example of a transient fault is a bit flip in the main memory. If an important data structure or critical instruction is stored in the affected memory location, it can lead to incorrect computations or program misbehavior. For instance, a bit flip in the memory storing a loop counter can cause the loop to

[2] Glitches: Momentary deviation in voltage, current, or signal, often causing incorrect operation.

[3] Combinational logic: Digital logic, wherein the output depends only on the current input states, not any past states.

execute indefinitely or terminate prematurely. Transient faults in control registers or flag bits can alter the flow of program execution, leading to unexpected jumps or incorrect branch decisions. In communication systems, transient faults can corrupt transmitted data packets, resulting in retransmissions or data loss.

In ML systems, transient faults can have significant implications during the training phase (Yi He et al. 2023). ML training involves iterative computations and updates to model parameters based on large datasets. If a transient fault occurs in the memory storing the model weights or gradients, it can lead to incorrect updates and compromise the convergence and accuracy of the training process. For example, a bit flip in the weight matrix of a neural network can cause the model to learn incorrect patterns or associations, leading to degraded performance (Wan et al. 2021). Transient faults in the data pipeline, such as corruption of training samples or labels, can also introduce noise and affect the quality of the learned model.

As shown in Figure 18.8, a real-world example from Google's production fleet highlights how an SDC anomaly caused a significant deviation in the gradient norm—a measure of the magnitude of updates to the model parameters. Such deviations can disrupt the optimization process, leading to slower convergence or failure to reach an optimal solution.

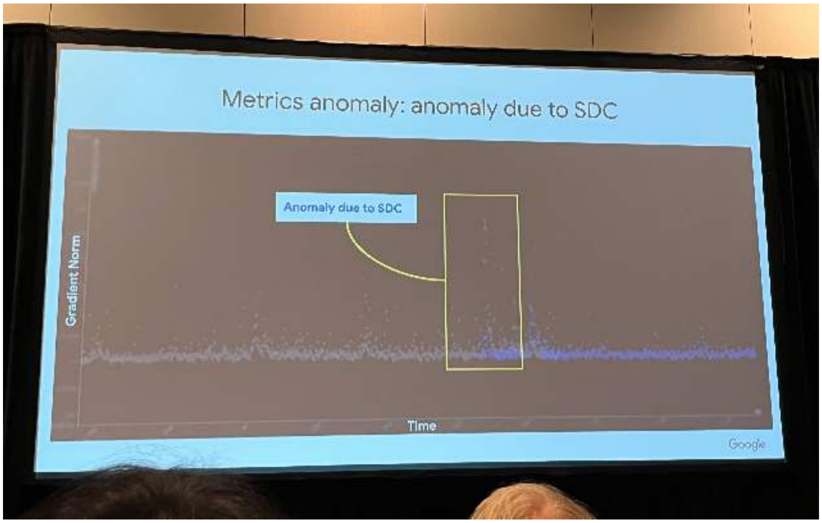

Metrics anomaly: anomaly due to SDC

Figure 18.8: SDC in ML training phase results in anomalies in the gradient norm. Source: Jeff Dean, MLSys 2024 Keynote (Google)

During the inference phase, transient faults can impact the reliability and trustworthiness of ML predictions. If a transient fault occurs in the memory storing the trained model parameters or during the computation of inference results, it can lead to incorrect or inconsistent predictions. For instance, a bit flip in the activation values of a neural network can alter the final classification or regression output (Mahmoud et al. 2020). In safety-critical applications, such as autonomous vehicles or medical diagnosis, these faults can have severe consequences, resulting in incorrect decisions or actions that may compromise safety or lead to system failures (G. Li et al. 2017; S. Jha et al. 2019).

Transient faults can be amplified in resource-constrained environments like TinyML, where limited computational and memory resources exacerbate their impact. One prominent example is Binarized Neural Networks (BNNs) (Courbariaux et al. 2016), which represent network weights in single-bit precision to achieve computational efficiency and faster inference times. While this binary representation is advantageous for resource-constrained systems, it also makes BNNs particularly fragile to bit-flip errors. For instance, prior work (Aygun, Gunes, and De Vleeschouwer 2021) has shown that a two-hidden-layer BNN architecture for a simple task such as MNIST classification suffers performance degradation from 98% test accuracy to 70% when random bit-flipping soft errors are inserted through model weights with a 10% probability. To address these vulnerabilities, techniques like flip-aware training and emerging approaches such as stochastic computing[4] are being explored to enhance fault tolerance.

18.3.2 Permanent Faults

Permanent faults are hardware defects that persist and cause irreversible damage to the affected components. These faults are characterized by their persistent nature and require repair or replacement of the faulty hardware to restore normal system functionality.

18.3.2.1 Characteristics

Permanent faults cause persistent and irreversible malfunctions in hardware components. The faulty component remains non-operational until it is repaired or replaced. These faults are consistent and reproducible, meaning the faulty behavior is observed every time the affected component is used. They can impact processors, memory modules, storage devices, or interconnects—potentially leading to system crashes, data corruption, or complete system failure.

One notable example of a permanent fault is the Intel FDIV bug, discovered in 1994. This flaw affected the floating-point division (FDIV) units of certain Intel Pentium processors, causing incorrect results for specific division operations and leading to inaccurate calculations.

The FDIV bug occurred due to an error in the lookup table[5] used by the division unit. In rare cases, the processor would fetch an incorrect value, resulting in a slightly less precise result than expected. For instance, Figure 18.9 shows a fraction 4195835/3145727 plotted on a Pentium processor with the FDIV fault. The triangular regions highlight where erroneous calculations occurred. Ideally, all correct values would round to 1.3338, but the faulty results showed 1.3337, indicating a mistake in the 5th digit.

Although the error was small, it could compound across many operations, significantly affecting results in precision-critical applications such as scientific simulations, financial calculations, and computer-aided design. The bug ultimately led to incorrect outcomes in these domains and underscored the severe consequences permanent faults can have.

The FDIV bug serves as a cautionary tale for ML systems. In such systems, permanent faults in hardware components can result in incorrect computations, impacting model accuracy and reliability. For example, if an ML system relies on a processor with a faulty floating-point unit, similar to the FDIV bug, it

[4] Stochastic Computing: A collection of techniques using random bits and logic operations to perform arithmetic and data processing, promising better fault tolerance.

[5] Lookup Table: A data structure used to replace a runtime computation with a simpler array indexing operation.

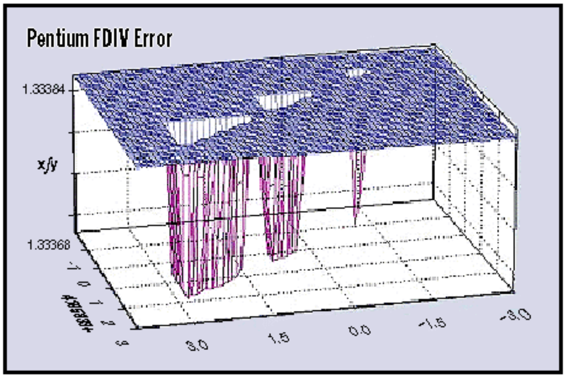


Figure 18.9: Intel Pentium processor with the FDIV permanent fault. The triangular regions are where erroneous calculations occurred. Source: Byte Magazine

could introduce persistent errors during training or inference. These errors may propagate through the model, leading to inaccurate predictions or skewed learning outcomes.

This is especially critical in safety-sensitive applications like autonomous driving, medical diagnosis, or financial forecasting, where the consequences of incorrect computations can be severe. ML practitioners must be aware of these risks and incorporate fault-tolerant techniques, including hardware redundancy, error detection and correction, and robust algorithm design, to mitigate them. Additionally, thorough hardware validation and testing can help identify and resolve permanent faults before they affect system performance and reliability.

18.3.2.2 Causes

Permanent faults can arise from two primary sources: manufacturing defects and wear-out mechanisms. Manufacturing defects are flaws introduced during the fabrication process, including improper etching, incorrect doping, or contamination. These defects may result in non-functional or partially functional components. In contrast, wear-out mechanisms occur over time due to prolonged use and operational stress. Phenomena like electromigration[6], oxide breakdown[7], and thermal stress[8] degrade component integrity, eventually leading to permanent failure.

18.3.2.3 Mechanisms

Permanent faults manifest through several mechanisms, depending on their nature and location. A common example is the stuck-at fault (Seong et al. 2010), where a signal or memory cell becomes permanently fixed at either 0 or 1, regardless of the intended input, as shown in Figure 18.10. This type of fault can occur in logic gates, memory cells, or interconnects and typically results in incorrect computations or persistent data corruption.

Other mechanisms include device failures, in which hardware components such as transistors or memory cells cease functioning entirely due to manufacturing defects or degradation over time. Bridging faults, which occur when two or more signal lines are unintentionally connected, can introduce short circuits or incorrect logic behaviors that are difficult to isolate.

[6] The movement of metal atoms in a conductor under the influence of an electric field.

[7] The failure of an oxide layer in a transistor due to excessive electric field stress.

[8] Degradation caused by repeated cycling through high and low temperatures.

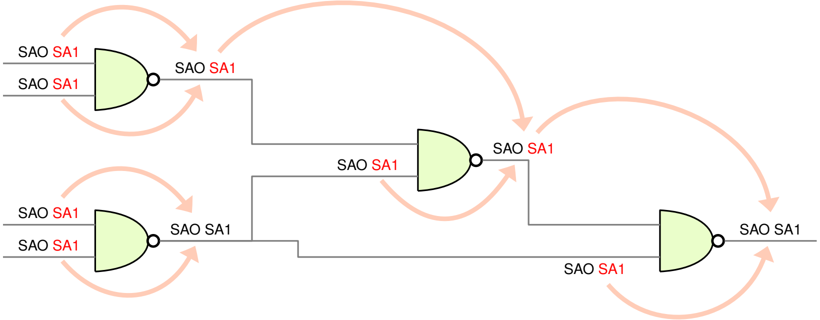


Figure 18.10: Stuck-at Fault Model in Digital Circuits. Source: Accendo Reliability

In more subtle cases, delay faults can arise when the propagation time of a signal exceeds the allowed timing constraints. Although the logical values may be correct, the violation of timing expectations can still result in erroneous behavior. Similarly, interconnect faults, including open circuits caused by broken connections, high-resistance paths that impede current flow, and increased capacitance that distorts signal transitions, can significantly degrade circuit performance and reliability.

Memory subsystems are particularly vulnerable to permanent faults. Transition faults can prevent a memory cell from successfully changing its state, while coupling faults result from unwanted interference between adjacent cells, leading to unintentional state changes. Additionally, neighborhood pattern sensitive faults occur when the state of a memory cell is incorrectly influenced by the data stored in nearby cells, reflecting a more complex interaction between circuit layout and logic behavior.

Finally, permanent faults can also occur in critical infrastructure components such as the power supply network or clock distribution system. Failures in these subsystems can affect circuit-wide functionality, introduce timing errors, or cause widespread operational instability.

Taken together, these mechanisms illustrate the varied and often complex ways in which permanent faults can undermine the behavior of computing systems. For ML applications in particular, where correctness and consistency are vital, understanding these fault modes is essential for developing resilient hardware and software solutions.

18.3.2.4 Impact on ML

Permanent faults can severely disrupt the behavior and reliability of computing systems. For example, a stuck-at fault in a processor's arithmetic logic unit (ALU) can produce persistent computational errors, leading to incorrect program behavior or crashes. In memory modules, such faults may corrupt stored data, while in storage devices, they can result in bad sectors or total data loss. Interconnect faults may interfere with data transmission, leading to system hangs or corruption.

For ML systems, these faults pose significant risks in both the training and inference phases. During training, permanent faults in processors or memory can lead to incorrect gradient calculations, corrupt model parameters, or prematurely halted training processes (Yi He et al. 2023). Similarly, faults in storage can compromise training datasets or saved models, affecting consistency and reliability.

In the inference phase, faults can distort prediction results or lead to runtime failures. For instance, errors in the hardware storing model weights might lead to outdated or corrupted models being used, while processor faults could yield incorrect outputs (J. J. Zhang et al. 2018).

To mitigate these impacts, ML systems must incorporate both hardware and software fault-tolerant techniques. Hardware-level methods include component redundancy and error-correcting codes (J. Kim, Sullivan, and Erez 2015).[9] Software approaches, like checkpoint and restart mechanisms[10] (Egwutuoha et al. 2013), allow systems to recover to a known-good state after a failure. Regular monitoring, testing, and maintenance can also help detect and replace failing components before critical errors occur.

Ultimately, designing ML systems with built-in fault tolerance is essential to ensure resilience. Incorporating redundancy, error-checking, and fail-safe mechanisms helps preserve model integrity, accuracy, and trustworthiness—even in the face of permanent hardware faults.

18.3.3 Intermittent Faults

Intermittent faults are hardware faults that occur sporadically and unpredictably in a system. An example is illustrated in Figure 18.11, where cracks in the material can introduce increased resistance in circuitry. These faults are particularly challenging to detect and diagnose because they appear and disappear intermittently, making it difficult to reproduce and isolate the root cause. Depending on their frequency and location, intermittent faults can lead to system instability, data corruption, and performance degradation.

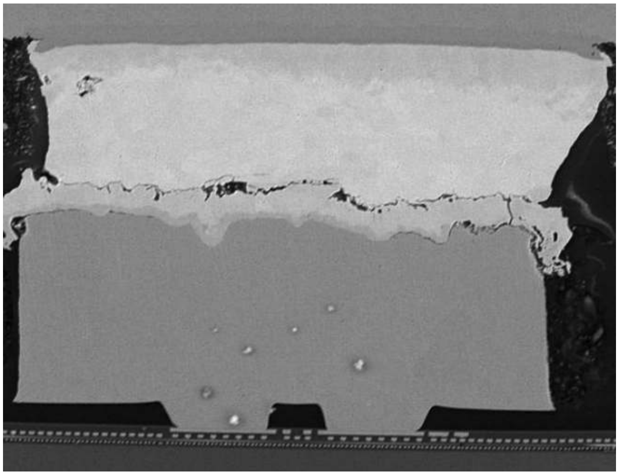


Figure 18.11: Increased resistance due to an intermittent fault – crack between copper bump and package solder. Source: Constantinescu

[9] Error-Correcting Codes: Methods used in data storage and transmission to detect and correct errors.

[10] Checkpoint and Restart Mechanisms: Techniques that periodically save a program's state so it can resume from the last saved state after a failure.

18.3.3.1 Characteristics

Intermittent faults are defined by their sporadic and non-deterministic behavior. They occur irregularly and may manifest for short durations, disappearing without a consistent pattern. Unlike permanent faults, they do not appear every time the affected component is used, which makes them particularly difficult to detect and reproduce. These faults can affect a variety of hardware components, including processors, memory modules, storage devices, and interconnects. As a result, they may lead to transient errors, unpredictable system behavior, or data corruption.

Their impact on system reliability can be significant. For instance, an intermittent fault in a processor's control logic may disrupt the normal execution path, causing irregular program flow or unexpected system hangs. In memory modules, such faults can alter stored values inconsistently, leading to errors that are difficult to trace. Storage devices affected by intermittent faults may suffer from sporadic read/write errors or data loss, while intermittent faults in communication channels can cause data corruption, packet loss, or unstable connectivity. Over time, these failures can accumulate, degrading system performance and reliability (Rashid, Pattabiraman, and Gopalakrishnan 2015).

18.3.3.2 Causes

The causes of intermittent faults are diverse, ranging from physical degradation to environmental influences. One common cause is the aging and wear-out of electronic components. As hardware endures prolonged operation, thermal cycling, and mechanical stress, it may develop cracks, fractures, or fatigue that introduce intermittent faults. For instance, solder joints in ball grid arrays (BGAs) or flip-chip packages can degrade over time, leading to intermittent open circuits or short circuits.

Manufacturing defects and process variations can also introduce marginal components that behave reliably under most circumstances but fail intermittently under stress or extreme conditions. For example, Figure 18.12 shows a residue-induced intermittent fault in a DRAM chip that leads to sporadic failures.

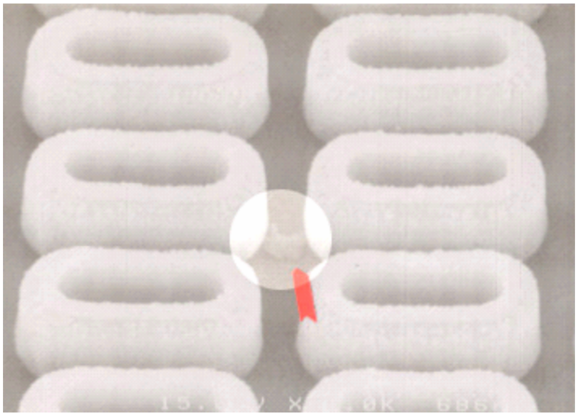


Figure 18.12: Residue induced intermittent fault in a DRAM chip. Source: Hynix Semiconductor

Environmental factors such as thermal cycling, humidity, mechanical vibrations, or electrostatic discharge can exacerbate these weaknesses and trigger faults that would not otherwise appear. Loose or degrading physical connections, including those found in connectors or printed circuit boards, are also common sources of intermittent failures, particularly in systems exposed to movement or temperature variation.

18.3.3.3 Mechanisms

Intermittent faults can manifest through various physical and logical mechanisms depending on their root causes. One such mechanism is the intermittent open or short circuit, where physical discontinuities or partial connections cause signal paths to behave unpredictably. These faults may momentarily disrupt signal integrity, leading to glitches or unexpected logic transitions.

Another common mechanism is the intermittent delay fault (J. Zhang et al. 2018), where signal propagation times fluctuate due to marginal timing conditions, resulting in synchronization issues and incorrect computations. In memory cells or registers, intermittent faults can appear as transient bit flips or soft errors, corrupting data in ways that are difficult to detect or reproduce. Because these faults are often condition-dependent, they may only emerge under specific thermal, voltage, or workload conditions, adding further complexity to their diagnosis.

18.3.3.4 Impact on ML

Intermittent faults pose significant challenges for ML systems by undermining computational consistency and model reliability. During the training phase, such faults in processing units or memory can cause sporadic errors in the computation of gradients, weight updates, or loss values. These errors may not be persistent but can accumulate across iterations, degrading convergence and leading to unstable or suboptimal models. Intermittent faults in storage may corrupt input data or saved model checkpoints, further affecting the training pipeline (Yi He et al. 2023).

In the inference phase, intermittent faults may result in inconsistent or erroneous predictions. Processing errors or memory corruption can distort activations, outputs, or intermediate representations of the model, particularly when faults affect model parameters or input data. Intermittent faults in data pipelines, such as unreliable sensors or storage systems, can introduce subtle input errors that degrade model robustness and output accuracy. In high-stakes applications like autonomous driving or medical diagnosis, these inconsistencies can result in dangerous decisions or failed operations.

Mitigating the effects of intermittent faults in ML systems requires a multi-layered approach (Rashid, Pattabiraman, and Gopalakrishnan 2012). At the hardware level, robust design practices, environmental controls, and the use of higher-quality or more reliable components can reduce susceptibility to fault conditions. Redundancy and error detection mechanisms can help identify and recover from transient manifestations of intermittent faults.

At the software level, techniques such as runtime monitoring, anomaly detection, and adaptive control strategies can provide resilience. Data validation checks, outlier detection, model ensembling, and runtime model adaptation are examples of fault-tolerant methods that can be integrated into ML pipelines to improve reliability in the presence of sporadic errors.

Ultimately, designing ML systems that can gracefully handle intermittent faults is essential to maintaining their accuracy, consistency, and dependability. This involves proactive fault detection, regular system monitoring, and ongoing maintenance to ensure early identification and remediation of issues. By embedding resilience into both the architecture and operational workflow, ML systems can remain robust even in environments prone to sporadic hardware failures.

18.3.4 Detection and Mitigation

Various fault detection techniques, including hardware-level and software-level approaches, and effective mitigation strategies can enhance the resilience of ML systems. Additionally, resilient ML system design considerations, case studies and examples, and future research directions in fault-tolerant ML systems provide insights into building robust systems.

18.3.4.1 Detection Techniques

Fault detection techniques are important for identifying and localizing hardware faults in ML systems. These techniques can be broadly categorized into hardware-level and software-level approaches, each offering unique capabilities and advantages.

Hardware-Level Detection. Hardware-level fault detection techniques are implemented at the physical level of the system and aim to identify faults in the underlying hardware components. There are several hardware techniques, but broadly, we can bucket these different mechanisms into the following categories.

Built-in self-test (BIST) Mechanisms. BIST is a powerful technique for detecting faults in hardware components (Bushnell and Agrawal 2002). It involves incorporating additional hardware circuitry into the system for self-testing and fault detection. BIST can be applied to various components, such as processors, memory modules, or application-specific integrated circuits (ASICs). For example, BIST can be implemented in a processor using scan chains[11], which are dedicated paths that allow access to internal registers and logic for testing purposes.

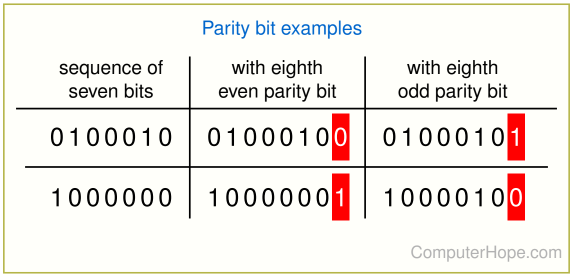
During the BIST process, predefined test patterns are applied to the processor's internal circuitry, and the responses are compared against expected values. Any discrepancies indicate the presence of faults. Intel's Xeon processors, for instance, include BIST mechanisms to test the CPU cores, cache memory, and other critical components during system startup.

Error Detection Codes. Error detection codes are widely used to detect data storage and transmission errors (Hamming 1950)[12]. These codes add redundant bits to the original data, allowing the detection of bit errors. Example: Parity checks are a simple form of error detection code shown in Figure 18.13[13]. In a single-bit parity scheme, an extra bit is appended to each data word, making the number of 1s in the word even (even parity) or odd (odd parity).

[11] Scan Chains: Dedicated paths incorporated within a processor that grant access to internal registers and logic for testing.

[12] R. W. Hamming's seminal paper introduced error detection and correction codes, significantly advancing digital communication reliability.

[13] In parity checks, an extra bit accounts for the total number of 1s in a data word, enabling fundamental error detection.



Parity bit examples

sequence of seven bits	with eighth even parity bit	with eighth odd parity bit
0100010	01000100	01000101
1000000	10000001	10000100

ComputerHope.com

Figure 18.13: Parity bit example. Source: Computer Hope

When reading the data, the parity is checked, and if it doesn't match the expected value, an error is detected. More advanced error detection codes, such as cyclic redundancy checks (CRC), calculate a checksum based on the data and append it to the message. The checksum is recalculated at the receiving end and compared with the transmitted checksum to detect errors. Error-correcting code (ECC) memory modules, commonly used in servers and critical systems, employ advanced error detection and correction codes to detect and correct single-bit or multi-bit errors in memory.

Hardware redundancy and voting mechanisms. Hardware redundancy involves duplicating critical components and comparing their outputs to detect and mask faults (Sheaffer, Luebke, and Skadron 2007). Voting mechanisms, such as double modular redundancy (DMR)[14] or triple modular redundancy (TMR)[15], employ multiple instances of a component and compare their outputs to identify and mask faulty behavior (Arifeen, Hassan, and Lee 2020).

In a DMR or TMR system, two or three identical instances of a hardware component, such as a processor or a sensor, perform the same computation in parallel. The outputs of these instances are fed into a voting circuit, which compares the results and selects the majority value as the final output. If one of the instances produces an incorrect result due to a fault, the voting mechanism masks the error and maintains the correct output. TMR is commonly used in aerospace and aviation systems, where high reliability is critical. For instance, the Boeing 777 aircraft employs TMR in its primary flight computer system to ensure the availability and correctness of flight control functions (Yeh, n.d.).

Tesla's self-driving computers, on the other hand, employ a DMR architecture to ensure the safety and reliability of critical functions such as perception, decision-making, and vehicle control, as shown in Figure 18.14. In Tesla's implementation, two identical hardware units, often called "redundant computers" or "redundant control units," perform the same computations in parallel. Each unit independently processes sensor data, executes algorithms, and generates control commands for the vehicle's actuators, such as steering, acceleration, and braking (Bannon et al. 2019).

The outputs of these two redundant units are continuously compared to detect any discrepancies or faults. If the outputs match, the system assumes that both units function correctly, and the control commands are sent to the vehicle's actuators. However, if there is a mismatch between the outputs, the

[14] Double Modular Redundancy (DMR): A fault-tolerance process in which computations are duplicated to identify and correct errors.

[15] Triple Modular Redundancy (TMR): A fault-tolerance process where three instances of a computation are performed to identify and correct errors.

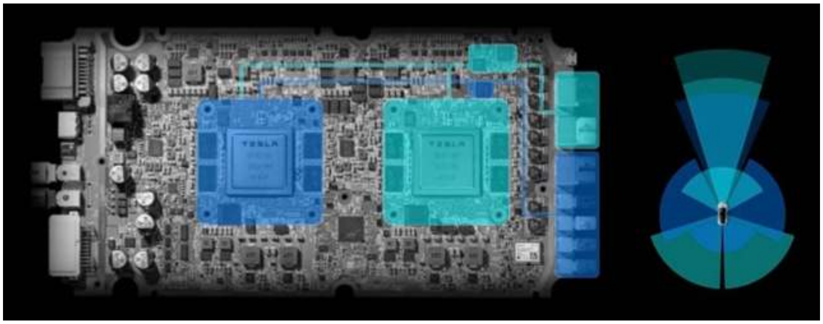


Figure 18.14: Tesla full self-driving computer with dual redundant SoCs. Source: Tesla

system identifies a potential fault in one of the units and takes appropriate action to ensure safe operation.

DMR in Tesla's self-driving computer provides an extra safety and fault tolerance layer. By having two independent units performing the same computations, the system can detect and mitigate faults that may occur in one of the units. This redundancy helps prevent single points of failure and ensures that critical functions remain operational despite hardware faults.

The system may employ additional mechanisms to determine which unit is faulty in a mismatch. This can involve using diagnostic algorithms, comparing the outputs with data from other sensors or subsystems, or analyzing the consistency of the outputs over time. Once the faulty unit is identified, the system can isolate it and continue operating using the output from the non-faulty unit.

Tesla also incorporates redundancy mechanisms beyond DMR. For example, they use redundant power supplies, steering and braking systems, and diverse sensor suites (e.g., cameras, radar, and ultrasonic sensors) to provide multiple layers of fault tolerance. These redundancies collectively contribute to the overall safety and reliability of the self-driving system.

It's important to note that while DMR provides fault detection and some level of fault tolerance, TMR may provide a different level of fault masking. In DMR, if both units experience simultaneous faults or the fault affects the comparison mechanism, the system may be unable to identify the fault. Therefore, Tesla's SDCs rely on a combination of DMR and other redundancy mechanisms to achieve a high level of fault tolerance.

The use of DMR in Tesla's self-driving computer highlights the importance of hardware redundancy in safety-critical applications. By employing redundant computing units and comparing their outputs, the system can detect and mitigate faults, enhancing the overall safety and reliability of the self-driving functionality.

Another approach to hardware redundancy is the use of hot spares[16], as employed by Google in its data centers to address SDC during ML training. Unlike DMR and TMR, which rely on parallel processing and voting mechanisms to detect and mask faults, hot spares provide fault tolerance by maintaining backup hardware units that can seamlessly take over computations when a fault is detected. As illustrated in Figure 18.15, during normal ML training, multiple

[16] Hot Spares: In a system redundancy design, these are the backup components kept ready to instantaneously replace failing components without disrupting the operation.

synchronous training workers process data in parallel. However, if a worker becomes defective and causes SDC, an SDC checker automatically identifies the issues. Upon detecting the SDC, the SDC checker moves the training to a hot spare and sends the defective machine for repair. This redundancy safeguards the continuity and reliability of ML training, effectively minimizing downtime and preserving data integrity.

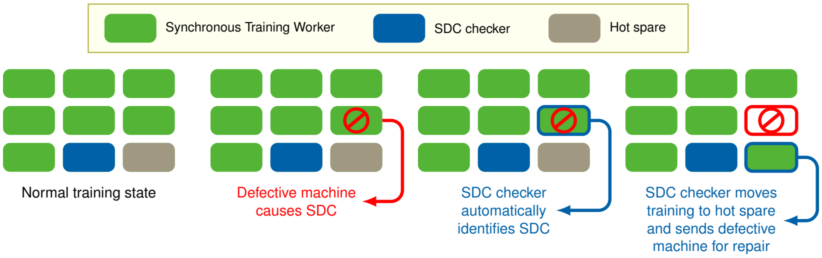

Synchronous Training Worker
SDC checker
Hot spare
Normal training state
Defective machine causes SDC
SDC checker automatically identifies SDC
SDC checker moves training to hot spare and sends defective machine for repair

Figure 18.15: Google employs hot spare cores to transparently handle SDCs in the data center. Source: Jeff Dean, MLSys 2024 Keynote (Google)

Watchdog timers. Watchdog timers are hardware components that monitor the execution of critical tasks or processes (Pont and Ong 2002). They are commonly used to detect and recover from software or hardware faults that cause a system to become unresponsive or stuck in an infinite loop. In an embedded system, a watchdog timer can be configured to monitor the execution of the main control loop, as illustrated in Figure 18.16. The software periodically resets the watchdog timer to indicate that it functions correctly. Suppose the software fails to reset the timer within a specified time limit (timeout period). In that case, the watchdog timer assumes that the system has encountered a fault and triggers a predefined recovery action, such as resetting the system or switching to a backup component. Watchdog timers are widely used in automotive electronics, industrial control systems, and other safety-critical applications to ensure the timely detection and recovery from faults.

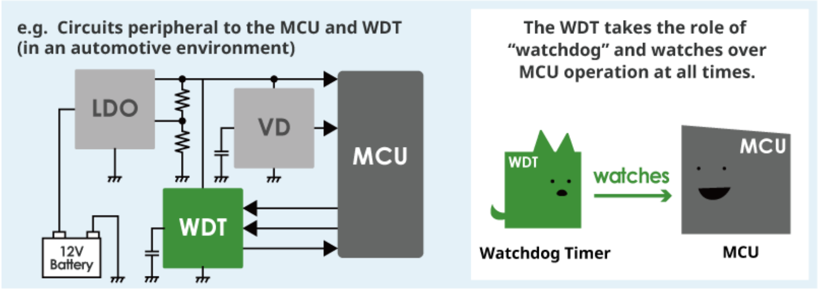

e.g. Circuits peripheral to the MCU and WDT (in an automotive environment)
LDO
VD
MCU
WDT
12V Battery
The WDT takes the role of "watchdog" and watches over MCU operation at all times.
WDT
watches
MCU
Watchdog Timer
MCU

Figure 18.16: Watchdog timer example in detecting MCU faults. Source: Ablic

Software-Level Detection. Software-level fault detection techniques rely on software algorithms and monitoring mechanisms to identify system faults. These techniques can be implemented at various levels of the software stack, including the operating system, middleware, or application level.

Runtime monitoring and anomaly detection. Runtime monitoring involves continuously observing the behavior of the system and its components during execution (Francalanza et al. 2017). It helps detect anomalies, errors, or unexpected behavior that may indicate the presence of faults. For example, consider an ML-based image classification system deployed in a self-driving car. Runtime monitoring can be implemented to track the classification model's performance and behavior (Mahmoud et al. 2021).

Anomaly detection algorithms can be applied to the model's predictions or intermediate layer activations, such as statistical outlier detection or machine learning-based approaches (e.g., One-Class SVM or Autoencoders) (Chandola, Banerjee, and Kumar 2009). Figure 18.17 shows example of anomaly detection. Suppose the monitoring system detects a significant deviation from the expected patterns, such as a sudden drop in classification accuracy or out-of-distribution samples. In that case, it can raise an alert indicating a potential fault in the model or the input data pipeline. This early detection allows for timely intervention and fault mitigation strategies to be applied.

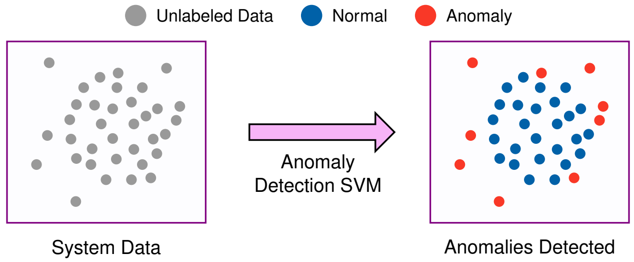


Figure 18.17: An example of anomaly detection using an SVM to analyze system logs and identify anomalies. Advanced methods, including unsupervised approaches, have been developed to enhance anomaly detection. Source: Google

Consistency checks and data validation. Consistency checks and data validation techniques ensure data integrity and correctness at different processing stages in an ML system (A. Lindholm et al. 2019). These checks help detect data corruption, inconsistencies, or errors that may propagate and affect the system's behavior. Example: In a distributed ML system where multiple nodes collaborate to train a model, consistency checks can be implemented to validate the integrity of the shared model parameters. Each node can compute a checksum or hash of the model parameters before and after the training iteration, as shown in Figure 18.17. Any inconsistencies or data corruption can be detected by comparing the checksums across nodes. Additionally, range checks can be applied to the input data and model outputs to ensure they fall within expected bounds. For instance, if an autonomous vehicle's perception system detects an object with unrealistic dimensions or velocities, it can indicate a fault in the sensor data or the perception algorithms (Wan et al. 2023).

Heartbeat and timeout mechanisms. Heartbeat mechanisms and timeouts are commonly used to detect faults in distributed systems and ensure the liveness and responsiveness of components (Kawazoe Aguilera, Chen, and Toueg 1997). These are quite similar to the watchdog timers found in hardware. For example, in a distributed ML system, where multiple nodes collaborate to perform tasks such as data preprocessing, model training, or inference, heartbeat mechanisms can be implemented to monitor the health and availability of each node. Each node periodically sends a heartbeat message to a central coordinator or its peer nodes, indicating its status and availability. Suppose a node fails to send a heartbeat within a specified timeout period, as shown in Figure 18.18. In that case, it is considered faulty, and appropriate actions can be taken, such as redistributing the workload or initiating a failover mechanism. Timeouts can also be used to detect and handle hanging or unresponsive components. For example, if a data loading process exceeds a predefined timeout threshold, it may indicate a fault in the data pipeline, and the system can take corrective measures.

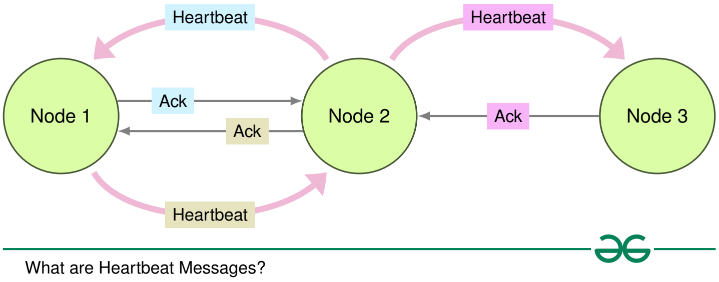


Figure 18.18: Heartbeat messages in distributed systems. Source: GeeksforGeeks

Software-implemented fault tolerance (SIFT) techniques. SIFT techniques introduce redundancy and fault detection mechanisms at the software level to improve the reliability and fault tolerance of the system (Reis et al., n.d.). Example: N-version programming is a SIFT technique where multiple functionally equivalent software component versions are developed independently by different teams. This can be applied to critical components such as the model inference engine in an ML system. Multiple versions of the inference engine can be executed in parallel, and their outputs can be compared for consistency. It is considered the correct result if most versions produce the same output. If there is a discrepancy, it indicates a potential fault in one or more versions, and appropriate error-handling mechanisms can be triggered. Another example is using software-based error correction codes, such as Reed-Solomon codes (Plank 1997), to detect and correct errors in data storage or transmission, as shown in Figure 18.19. These codes add redundancy to the data, enabling detecting and correcting certain errors and enhancing the system's fault tolerance.

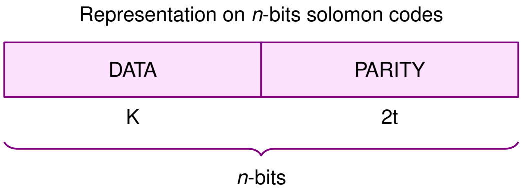


Figure 18.19: n-bits representation of the Reed-Solomon codes. Source: GeeksforGeeks

Caution 1: Anomaly Detection

In this Colab, play the role of an AI fault detective! You'll build an autoencoder-based anomaly detector to pinpoint errors in heart health data. Learn how to identify malfunctions in ML systems, a vital skill for creating dependable AI. We'll use Keras Tuner to fine-tune your autoencoder for top-notch fault detection. This experience directly links to the Robust AI chapter, demonstrating the importance of fault detection in real-world applications like healthcare and autonomous systems. Get ready to strengthen the reliability of your AI creations!

Open in Colab

18.3.5 Summary

Table 18.1 provides a comparative analysis of transient, permanent, and intermittent faults. It outlines the primary characteristics or dimensions that distinguish these fault types. Here, we summarize the relevant dimensions we examined and explore the nuances that differentiate transient, permanent, and intermittent faults in greater detail.

Table 18.1: Comparison of transient, permanent, and intermittent faults.

Dimension	Transient Faults	Permanent Faults	Intermittent Faults
Duration	Short-lived, temporary	Persistent, remains until repair or replacement	Sporadic, appears and disappears intermittently
Persistence	Disappears after the fault condition passes	Consistently present until addressed	Recurs irregularly, not always present
Causes	External factors (e.g., electromagnetic interference cosmic rays)	Hardware defects, physical damage, wear-out	Unstable hardware conditions, loose connections, aging components
Manifestation	Bit flips, glitches, temporary data corruption	Stuck-at faults, broken components, complete device failures	Occasional bit flips, intermittent signal issues, sporadic malfunctions
Impact on ML Systems	Introduces temporary errors or noise in computations	Causes consistent errors or failures, affecting reliability	Leads to sporadic and unpredictable errors, challenging to diagnose and mitigate
Detection	Error detection codes, comparison with expected values	Built-in self-tests, error detection codes, consistency checks	Monitoring for anomalies, analyzing error patterns and correlations
Mitigation	Error correction codes, redundancy, checkpoint and restart	Hardware repair or replacement, component redundancy, failover mechanisms	Robust design, environmental control, runtime monitoring, fault-tolerant techniques

18.4 Model Robustness

18.4.1 Adversarial Attacks

We first introduced adversarial attacks when discussing how slight changes to input data can trick a model into making incorrect predictions. These attacks often involve adding small, carefully designed perturbations to input data, which can cause the model to misclassify it, as shown in Figure 18.20. In this section, we will look at the different types of adversarial attacks and their impact on machine learning models. Understanding these attacks highlights why it is important to build models that are robust and able to handle these kinds of challenges.

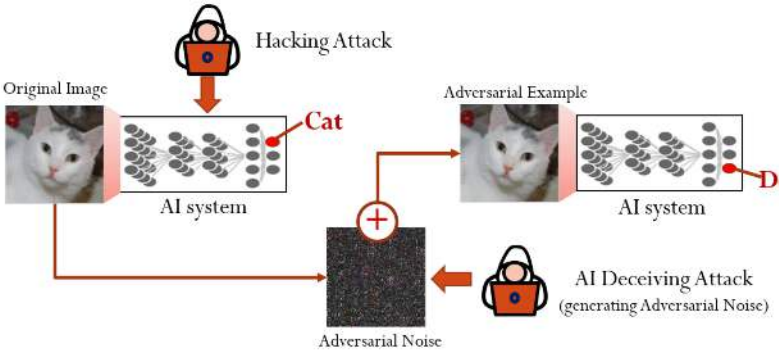


Figure 18.20: A small adversarial noise added to the original image can make the neural network classify the image as a Guacamole instead of an Egyptian cat. Source: Sutanto

18.4.1.1 Mechanisms

Gradient-based Attacks. One prominent category of adversarial attacks is gradient-based attacks. These attacks leverage the gradients of the ML model's loss function to craft adversarial examples. The Fast Gradient Sign Method (FGSM) is a well-known technique in this category. FGSM perturbs the input data by adding small noise in the direction of the gradient of the loss with respect to the input. The goal is to maximize the model's prediction error with minimal distortion to the original input.

The adversarial example is generated using the following formula:

$$x_{\text{adv}} = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

Where:

- x is the original input,
- y is the true label,
- θ represents the model parameters,
- $J(\theta, x, y)$ is the loss function,
- ϵ is a small scalar that controls the magnitude of the perturbation.

This method allows for fast and efficient generation of adversarial examples by taking a single step in the direction that increases the loss most rapidly, as shown in Figure 18.21.

Another variant, the Projected Gradient Descent (PGD) attack, extends FGSM by iteratively applying the gradient update step, allowing for more refined and powerful adversarial examples. PGD projects each perturbation step back into a constrained norm ball around the original input, ensuring that the adversarial example remains within a specified distortion limit. This makes PGD a stronger white-box attack and a benchmark for evaluating model robustness.

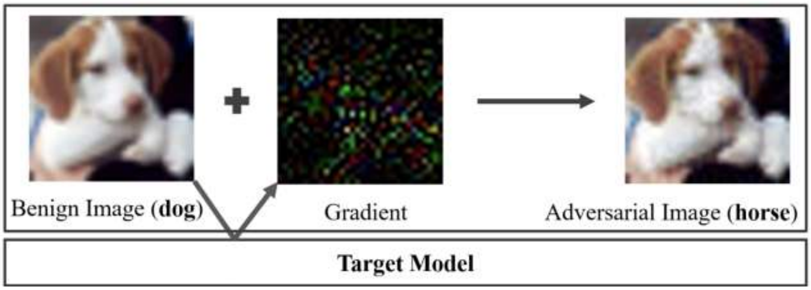


Figure 18.21: Gradient-Based Attacks. Source: Ivezic

The Jacobian-based Saliency Map Attack (JSMA) is another gradient-based approach that identifies the most influential input features and perturbs them to create adversarial examples. By constructing a saliency map based on the Jacobian of the model's outputs with respect to inputs, JSMA selectively alters a small number of input dimensions that are most likely to influence the target class. This makes JSMA more precise and targeted than FGSM or PGD, often requiring fewer perturbations to fool the model.

Gradient-based attacks are particularly effective in white-box settings, where the attacker has access to the model's architecture and gradients. Their efficiency and relative simplicity have made them popular tools for both attacking and evaluating model robustness in research.

Optimization-based Attacks. These attacks formulate the generation of adversarial examples as an optimization problem. The Carlini and Wagner (C&W) attack is a prominent example in this category. It finds the smallest perturbation that can cause misclassification while maintaining the perceptual similarity to the original input. The C&W attack employs an iterative optimization process to minimize the perturbation while maximizing the model's prediction error. It uses a customized loss function with a confidence term to generate more confident misclassifications.

C&W attacks are especially difficult to detect because the perturbations are typically imperceptible to humans, and they often bypass many existing defenses. The attack can be formulated under various norm constraints (e.g., L2, L∞) depending on the desired properties of the adversarial perturbation.

Another optimization-based approach is the Elastic Net Attack to DNNs (EAD), which incorporates elastic net regularization (a combination of L1 and L2 penalties) to generate adversarial examples with sparse perturbations. This can lead to minimal and localized changes in the input, which are harder to identify and filter. EAD is particularly useful in settings where perturbations need to be constrained in both magnitude and spatial extent.

These attacks are more computationally intensive than gradient-based methods but offer finer control over the adversarial example's properties. They are often used in high-stakes domains where stealth and precision are critical.

Transfer-based Attacks. Transfer-based attacks exploit the transferability property of adversarial examples. Transferability refers to the phenomenon where adversarial examples crafted for one ML model can often fool other models, even if they have different architectures or were trained on different datasets. This enables attackers to generate adversarial examples using a surrogate model and then transfer them to the target model without requiring direct access to its parameters or gradients.

This property underlies the feasibility of black-box attacks, where the adversary cannot query gradients but can still fool a model by crafting attacks on a publicly available or similar substitute model. Transfer-based attacks are particularly relevant in practical threat scenarios, such as attacking commercial ML APIs, where the attacker can observe inputs and outputs but not internal computations.

Attack success often depends on factors like similarity between models, alignment in training data, and the regularization techniques used. Techniques like input diversity (random resizing, cropping) and momentum during optimization can be used to increase transferability.

Physical-world Attacks. Physical-world attacks bring adversarial examples into the realm of real-world scenarios. These attacks involve creating physical objects or manipulations that can deceive ML models when captured by sensors or cameras. Adversarial patches, for example, are small, carefully designed patterns that can be placed on objects to fool object detection or classification models. These patches are designed to work under varying lighting conditions, viewing angles, and distances, making them robust in real-world environments.

When attached to real-world objects, such as a stop sign or a piece of clothing, these patches can cause models to misclassify or fail to detect the objects accurately. Notably, the effectiveness of these attacks persists even after being printed out and viewed through a camera lens, bridging the digital and physical divide in adversarial ML.

Adversarial objects, such as 3D-printed sculptures or modified road signs, can also be crafted to deceive ML systems in physical environments. For example, a 3D turtle object was shown to be consistently classified as a rifle by an image classifier, even when viewed from different angles. These attacks underscore the risks facing AI systems deployed in physical spaces, such as autonomous vehicles, drones, and surveillance systems.

Research into physical-world attacks also includes efforts to develop universal adversarial perturbations—perturbations that can fool a wide range of inputs and models. These threats raise serious questions about safety, robustness, and generalization in AI systems.

Summary. Table 18.2 provides a concise overview of the different categories of adversarial attacks, including gradient-based attacks (FGSM, PGD, JSMA), optimization-based attacks (C&W, EAD), transfer-based attacks, and physical-world attacks (adversarial patches and objects). Each attack is briefly described, highlighting its key characteristics and mechanisms.

Table 18.2: Different attack types on ML models.

Attack Category	Attack Name	Description
Gradient-based	Fast Gradient Sign Method (FGSM) Projected Gradient Descent (PGD) Jacobian-based Saliency Map Attack (JSMA)	Perturbs input data by adding small noise in the gradient direction to maximize prediction error. Extends FGSM by iteratively applying the gradient update step for more refined adversarial examples. Identifies influential input features and perturbs them to create adversarial examples.
Optimization-based	Carlini and Wagner (C&W) Attack Elastic Net Attack to DNNs (EAD)	Finds the smallest perturbation that causes misclassification while maintaining perceptual similarity. Incorporates elastic net regularization to generate adversarial examples with sparse perturbations.
Transfer-based	Transferability-based Attacks	Exploits the transferability of adversarial examples across different models, enabling black-box attacks.
Physical-world	Adversarial Patches Adversarial Objects	Small, carefully designed patches placed on objects to fool object detection or classification models. Physical objects (e.g., 3D-printed sculptures, modified road signs) crafted to deceive ML systems in real-world scenarios.

The mechanisms of adversarial attacks reveal the intricate interplay between the ML model's decision boundaries, the input data, and the attacker's objectives. By carefully manipulating the input data, attackers can exploit the model's sensitivities and blind spots, leading to incorrect predictions. The success of adversarial attacks highlights the need for a deeper understanding of ML models' robustness and generalization properties.

Defending against adversarial attacks requires a multifaceted approach. Adversarial training is one common defense strategy in which models are trained on adversarial examples to improve robustness. Exposing the model to adversarial examples during training teaches it to classify them correctly and become more resilient to attacks. Defensive distillation, input preprocessing, and ensemble methods are other techniques that can help mitigate the impact of adversarial attacks.

As adversarial machine learning evolves, researchers explore new attack mechanisms and develop more sophisticated defenses. The arms race between attackers and defenders drives the need for constant innovation and vigilance in securing ML systems against adversarial threats. Understanding the mechanisms of adversarial attacks is crucial for developing robust and reliable ML models that can withstand the ever-evolving landscape of adversarial examples.

18.4.1.2 Impact on ML

Adversarial attacks on machine learning systems have emerged as a significant concern in recent years, highlighting the potential vulnerabilities and risks associated with the widespread adoption of ML technologies. These attacks involve carefully crafted perturbations to input data that can deceive or mislead ML models, leading to incorrect predictions or misclassifications, as shown in Figure 18.22. The impact of adversarial attacks on ML systems is far-reaching and can have serious consequences in various domains.

One striking example of the impact of adversarial attacks was demonstrated by researchers in 2017. They experimented with small black and white stickers on stop signs (Eykholt et al. 2017). To the human eye, these stickers did not obscure the sign or prevent its interpretability. However, when images of the

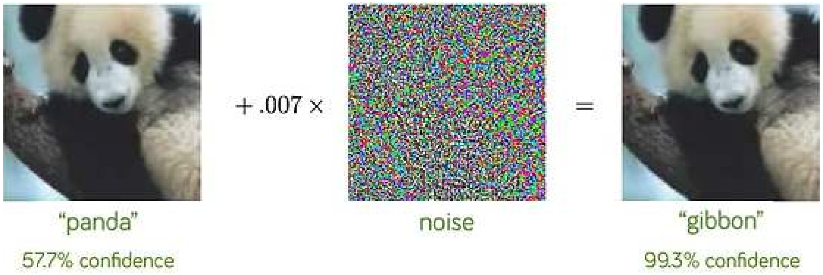


Figure 18.22: Adversarial example generation applied to GoogLeNet (Szegedy et al., 2014a) on ImageNet. Source: Goodfellow

sticker-modified stop signs were fed into standard traffic sign classification ML models, a shocking result emerged. The models misclassified the stop signs as speed limit signs over 85% of the time.

This demonstration shed light on the alarming potential of simple adversarial stickers to trick ML systems into misreading critical road signs. The implications of such attacks in the real world are significant, particularly in the context of autonomous vehicles. If deployed on actual roads, these adversarial stickers could cause self-driving cars to misinterpret stop signs as speed limits, leading to dangerous situations, as shown in Figure 18.23. Researchers warned that this could result in rolling stops or unintended acceleration into intersections, endangering public safety.

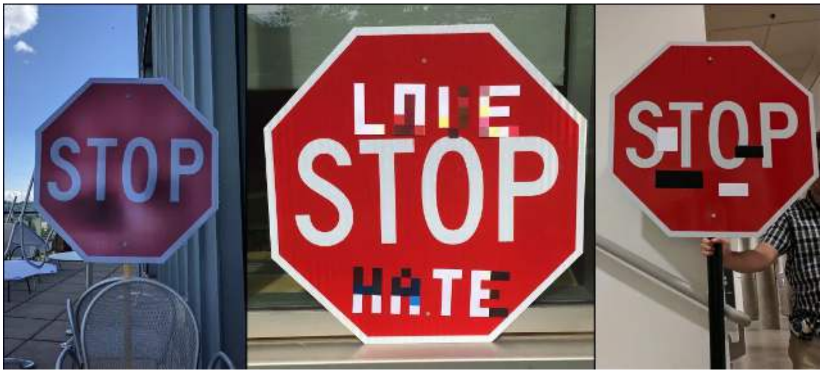


Figure 18.23: Graffiti on a stop sign tricked a self-driving car into thinking it was a 45 mph speed limit sign. Source: Eykholt

The case study of the adversarial stickers on stop signs provides a concrete illustration of how adversarial examples exploit how ML models recognize patterns. By subtly manipulating the input data in ways that are invisible to humans, attackers can induce incorrect predictions and create serious risks, especially in safety-critical applications like autonomous vehicles. The attack's simplicity highlights the vulnerability of ML models to even minor changes in the input, emphasizing the need for robust defenses against such threats.

The impact of adversarial attacks extends beyond the degradation of model performance. These attacks raise significant security and safety concerns, particularly in domains where ML models are relied upon for critical decision-making. In healthcare applications, adversarial attacks on medical imaging models could lead to misdiagnosis or incorrect treatment recommendations, jeopardizing patient well-being (M.-J. Tsai, Lin, and Lee 2023). In financial systems, adversarial attacks could enable fraud or manipulation of trading algorithms, resulting in substantial economic losses.

Moreover, adversarial vulnerabilities undermine the trustworthiness and interpretability of ML models. If carefully crafted perturbations can easily fool models, confidence in their predictions and decisions erodes. Adversarial examples expose the models' reliance on superficial patterns and the inability to capture the true underlying concepts, challenging the reliability of ML systems (Fursov et al. 2021).

Defending against adversarial attacks often requires additional computational resources and can impact the overall system performance. Techniques like adversarial training, where models are trained on adversarial examples to improve robustness, can significantly increase training time and computational requirements (Bai et al. 2021). Runtime detection and mitigation mechanisms, such as input preprocessing (Addepalli et al. 2020) or prediction consistency checks, introduce latency and affect the real-time performance of ML systems.

The presence of adversarial vulnerabilities also complicates the deployment and maintenance of ML systems. System designers and operators must consider the potential for adversarial attacks and incorporate appropriate defenses and monitoring mechanisms. Regular updates and retraining of models become necessary to adapt to new adversarial techniques and maintain system security and performance over time.

The impact of adversarial attacks on ML systems is significant and multifaceted. These attacks expose ML models' vulnerabilities, from degrading model performance and raising security and safety concerns to challenging model trustworthiness and interpretability. Developers and researchers must prioritize the development of robust defenses and countermeasures to mitigate the risks posed by adversarial attacks. By addressing these challenges, we can build more secure, reliable, and trustworthy ML systems that can withstand the ever-evolving landscape of adversarial threats.

18.4.2 Data Poisoning

Data poisoning presents a critical challenge to the integrity and reliability of machine learning systems. By introducing carefully crafted malicious data into the training pipeline, adversaries can subtly manipulate model behavior in ways that are difficult to detect through standard validation procedures. Unlike adversarial examples, which target models at inference time, poisoning attacks exploit upstream components of the system—such as data collection, labeling, or ingestion. As ML systems are increasingly deployed in automated and high-stakes environments, understanding how poisoning occurs and how it propagates through the system is essential for developing effective defenses.

18.4.2.1 Characteristics

Data poisoning is an attack in which the training data is deliberately manipulated to compromise the performance or behavior of a machine learning model, as described in (Biggio, Nelson, and Laskov 2012) and illustrated in Figure 18.24.

Attackers may alter existing training samples, introduce malicious examples, or interfere with the data collection pipeline. The result is a model that learns biased, inaccurate, or exploitable patterns.

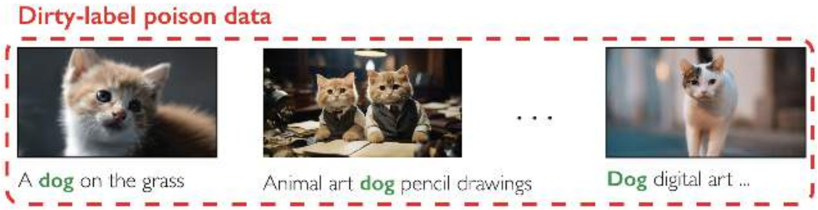


Figure 18.24: Samples of dirty-label poison data regarding mismatched text/image pairs. Source: (Shan et al. 2023)

In most cases, data poisoning unfolds in three stages.

In the injection stage, the attacker introduces poisoned samples into the training dataset. These samples may be altered versions of existing data or entirely new instances designed to blend in with clean examples. While they appear benign on the surface, these inputs are engineered to influence model behavior in subtle but deliberate ways. The attacker may target specific classes, insert malicious triggers, or craft outliers intended to distort the decision boundary.

During the training phase, the machine learning model incorporates the poisoned data and learns spurious or misleading patterns. These learned associations may bias the model toward incorrect classifications, introduce vulnerabilities, or embed backdoors. Because the poisoned data is often statistically similar to clean data, the corruption process typically goes unnoticed during standard model training and evaluation.

Finally, in the deployment stage, the attacker leverages the compromised model for malicious purposes. This could involve triggering specific behaviors, including the misclassification of an input that contains a hidden pattern, or simply exploiting the model's degraded accuracy in production. In real-world systems, such attacks can be difficult to trace back to training data, especially if the system's behavior appears erratic only in edge cases or under adversarial conditions.

The consequences of such manipulation are especially severe in high-stakes domains like healthcare, where even small disruptions to training data can lead to dangerous misdiagnoses or loss of trust in AI-based systems (Marulli, Marrone, and Verde 2022).

Four main categories of poisoning attacks have been identified in the literature (Oprea, Singhal, and Vassilev 2022). In availability attacks, a substantial portion of the training data is poisoned with the aim of degrading overall model performance. A classic example involves flipping labels—for instance, systematically changing instances with true label $y = 1$ to $y = 0$ in a binary classification task. These attacks render the model unreliable across a wide range of inputs, effectively making it unusable.

In contrast, targeted poisoning attacks aim to compromise only specific classes or instances. Here, the attacker modifies just enough data to cause a small set of inputs to be misclassified, while overall accuracy remains relatively stable. This subtlety makes targeted attacks especially hard to detect.

Backdoor poisoning introduces hidden triggers into training data—subtle patterns or features that the model learns to associate with a particular output. When the trigger appears at inference time, the model is manipulated into producing a predetermined response. These attacks are often effective even if the trigger pattern is imperceptible to human observers.

Subpopulation poisoning focuses on compromising a specific subset of the data population. While similar in intent to targeted attacks, subpopulation poisoning applies availability-style degradation to a localized group, for example, a particular demographic or feature cluster, while leaving the rest of the model's performance intact. This distinction makes such attacks both highly effective and especially dangerous in fairness-sensitive applications.

A common thread across these poisoning strategies is their subtlety. Manipulated samples are typically indistinguishable from clean data, making them difficult to identify through casual inspection or standard data validation. These manipulations might involve small changes to numeric values, slight label inconsistencies, or embedded visual patterns—each designed to blend into the data distribution while still affecting model behavior.

Such attacks may be carried out by internal actors, like data engineers or annotators with privileged access, or by external adversaries who exploit weak points in the data collection pipeline. In crowdsourced environments or open data collection scenarios, poisoning can be as simple as injecting malicious samples into a shared dataset or influencing user-generated content.

Crucially, poisoning attacks often target the early stages of the ML pipeline, such as collection and preprocessing, where there may be limited oversight. If data is pulled from unverified sources or lacks strong validation protocols, attackers can slip in poisoned data that appears statistically normal. The absence of integrity checks, robust outlier detection, or lineage tracking only heightens the risk.

Ultimately, the goal of these attacks is to corrupt the learning process itself. A model trained on poisoned data may learn spurious correlations, overfit to false signals, or become vulnerable to highly specific exploit conditions. Whether the result is a degraded model or one with a hidden exploit path, the trustworthiness and safety of the system are fundamentally compromised.

18.4.2.2 Mechanisms

Data poisoning can be implemented through a variety of mechanisms, depending on the attacker's access to the system and understanding of the data pipeline. These mechanisms reflect different strategies for how the training data can be corrupted to achieve malicious outcomes.

One of the most direct approaches involves modifying the labels of training data. In this method, an attacker selects a subset of training samples and alters their labels—flipping $y = 1$ to $y = 0$, or reassigning categories in multi-class settings. As shown in Figure 18.25, even small-scale label inconsistencies can lead to significant distributional shifts and learning disruptions.

Another mechanism involves modifying the input features of training examples without changing the labels. This might include imperceptible pixel-level changes in images, subtle perturbations in structured data, or embedding fixed

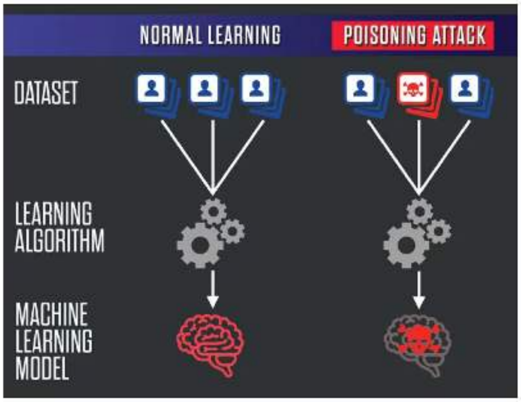


Figure 18.25: Garbage In – Garbage Out. Source: (Shan et al. 2023)

patterns that act as triggers for backdoor attacks. These alterations are often designed using optimization techniques that maximize their influence on the model while minimizing detectability.

More sophisticated attacks generate entirely new, malicious training examples. These synthetic samples may be created using adversarial methods, generative models, or even data synthesis tools. The aim is to carefully craft inputs that will distort the decision boundary of the model when incorporated into the training set. Such inputs may appear natural and legitimate but are engineered to introduce vulnerabilities.

Other attackers focus on weaknesses in data collection and preprocessing. If the training data is sourced from web scraping, social media, or untrusted user submissions, poisoned samples can be introduced upstream. These samples may pass through insufficient cleaning or validation checks, reaching the model in a "trusted" form. This is particularly dangerous in automated pipelines where human review is limited or absent.

In physically deployed systems, attackers may manipulate data at the source—for example, altering the environment captured by a sensor. A self-driving car might encounter poisoned data if visual markers on a road sign are subtly altered, causing the model to misclassify it during training. This kind of environmental poisoning blurs the line between adversarial attacks and data poisoning, but the mechanism, which involves compromising the training data, is the same.

Online learning systems represent another unique attack surface. These systems continuously adapt to new data streams, making them particularly susceptible to gradual poisoning. An attacker may introduce malicious samples incrementally, causing slow but steady shifts in model behavior. This form of attack is illustrated in Figure 18.26.

Insider collaboration adds a final layer of complexity. Malicious actors with legitimate access to training data, including annotators, researchers, or data vendors, can craft poisoning strategies that are more targeted and subtle than external attacks. These insiders may have knowledge of the model architec-

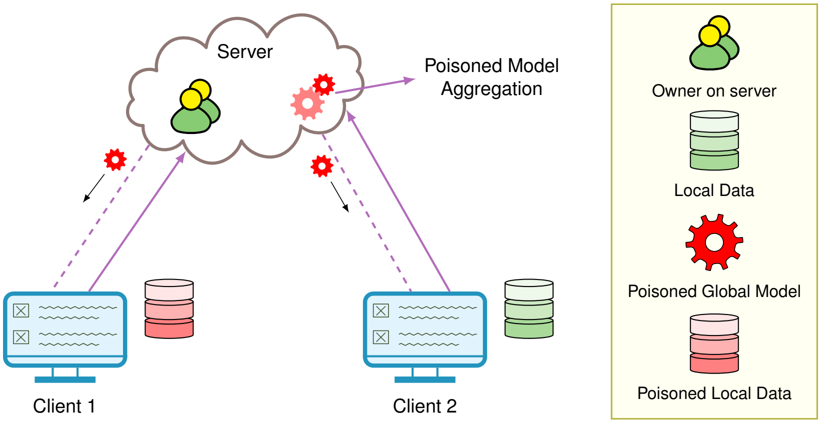


Figure 18.26: Data Poisoning Attack.

ture or training procedures, giving them an advantage in designing effective poisoning schemes.

Defending against these diverse mechanisms requires a multi-pronged approach: secure data collection protocols, anomaly detection, robust preprocessing pipelines, and strong access control. Validation mechanisms must be sophisticated enough to detect not only outliers but also cleverly disguised poisoned samples that sit within the statistical norm.

18.4.2.3 Impact on ML

The effects of data poisoning extend far beyond simple accuracy degradation. In the most general sense, a poisoned dataset leads to a corrupted model. But the specific consequences depend on the attack vector and the adversary's objective.

One common outcome is the degradation of overall model performance. When large portions of the training set are poisoned, often through label flipping or the introduction of noisy features, the model struggles to identify valid patterns, leading to lower accuracy, recall, or precision. In mission-critical applications like medical diagnosis or fraud detection, even small performance losses can result in significant real-world harm.

Targeted poisoning presents a different kind of danger. Rather than undermining the model's general performance, these attacks cause specific misclassifications. A malware detector, for instance, may be engineered to ignore one particular signature, allowing a single attack to bypass security. Similarly, a facial recognition model might be manipulated to misidentify a specific individual, while functioning normally for others.

Some poisoning attacks introduce hidden vulnerabilities in the form of backdoors or trojans. These poisoned models behave as expected during evaluation but respond in a malicious way when presented with specific triggers. In such cases, attackers can "activate" the exploit on demand, bypassing system protections without triggering alerts.

Bias is another insidious impact of data poisoning. If an attacker poisons samples tied to a specific demographic or feature group, they can skew the model's outputs in biased or discriminatory ways. Such attacks threaten fairness, amplify existing societal inequities, and are difficult to diagnose if the overall model metrics remain high.

Ultimately, data poisoning undermines the trustworthiness of the system itself. A model trained on poisoned data cannot be considered reliable, even if it performs well in benchmark evaluations. This erosion of trust has profound implications, particularly in fields like autonomous systems, financial modeling, and public policy.

18.4.2.4 Case Study: Art Protection via Poisoning

Interestingly, not all data poisoning is malicious. Researchers have begun to explore its use as a defensive tool, particularly in the context of protecting creative work from unauthorized use by generative AI models.

A compelling example is Nightshade, developed by researchers at the University of Chicago to help artists prevent their work from being scraped and used to train image generation models without consent (Shan et al. 2023). Nightshade allows artists to apply subtle perturbations to their images before publishing them online. These changes are invisible to human viewers but cause serious degradation in generative models that incorporate them into training.

When Stable Diffusion was trained on just 300 poisoned images, the model began producing bizarre outputs—such as cows when prompted with "car," or cat-like creatures in response to "dog." These results, visualized in Figure 18.27, show how effectively poisoned samples can distort a model's conceptual associations.

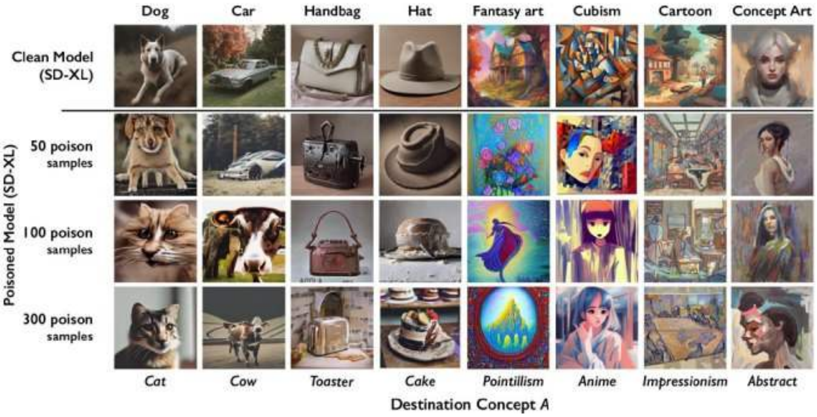


Figure 18.27: NightShade's poisoning effects on Stable Diffusion. Source: (Shan et al. 2023)

What makes Nightshade especially potent is the cascading effect of poisoned concepts. Because generative models rely on semantic relationships between categories, a poisoned "car" can bleed into related concepts like "truck," "bus," or "train," leading to widespread hallucinations.

However, like any powerful tool, Nightshade also introduces risks. The same technique used to protect artistic content could be repurposed to sabotage legitimate training pipelines, highlighting the dual-use dilemma[17] at the heart

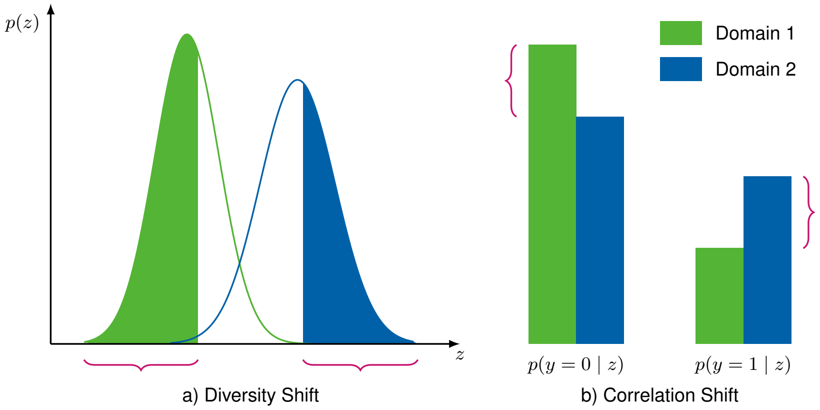
[17] Dual-use Dilemma: In AI, the challenge of mitigating misuse of technology that has both positive and negative potential uses.

of modern machine learning security.

18.4.3 Distribution Shifts

18.4.3.1 Characteristics

Distribution shift refers to the phenomenon where the data distribution encountered by a machine learning model during deployment differs from the distribution it was trained on, as shown in Figure 18.28. This change in distribution is not necessarily the result of a malicious attack. Rather, it often reflects the natural evolution of real-world environments over time. In essence, the statistical properties, patterns, or assumptions in the data may change between training and inference phases, which can lead to unexpected or degraded model performance.



a) Diversity Shift

b) Correlation Shift

Figure 18.28: The curly brackets enclose the distribution shift between the environments. Here, z stands for the spurious feature, and y stands for label class.

A distribution shift typically takes one of several forms:

- **Covariate shift**, where the input distribution $P(x)$ changes while the conditional label distribution $P(y \mid x)$ remains stable.
- **Label shift**, where the label distribution $P(y)$ changes while $P(x \mid y)$ stays the same.
- **Concept drift**, where the relationship between inputs and outputs—$P(y \mid x)$—evolves over time.
- **Concept drift**, where the relationship between inputs and outputs, $P(y \mid x)$, evolves over time.

These formal definitions help frame more intuitive examples of shift that are commonly encountered in practice.

One of the most common causes is domain mismatch, where the model is deployed on data from a different domain than it was trained on. For example, a sentiment analysis model trained on movie reviews may perform poorly when applied to tweets, due to differences in language, tone, and structure. In this case, the model has learned domain-specific features that do not generalize well to new contexts.

Another major source is temporal drift, where the input distribution evolves gradually or suddenly over time. In production settings, data changes due to new trends, seasonal effects, or shifts in user behavior. For instance, in a fraud detection system, fraud patterns may evolve as adversaries adapt. Without ongoing monitoring or retraining, models become stale and ineffective. This form of shift is visualized in Figure 18.29.

Contextual changes arise when deployment environments differ from training conditions due to external factors such as lighting, sensor variation, or user behavior. For example, a vision model trained in a lab under controlled lighting may underperform when deployed in outdoor or dynamic environments.

Another subtle but critical factor is unrepresentative training data. If the training dataset fails to capture the full variability of the production environment, the model may generalize poorly. For example, a facial recognition model trained predominantly on one demographic group may produce biased or inaccurate predictions when deployed more broadly. In this case, the shift reflects missing diversity or structure in the training data.

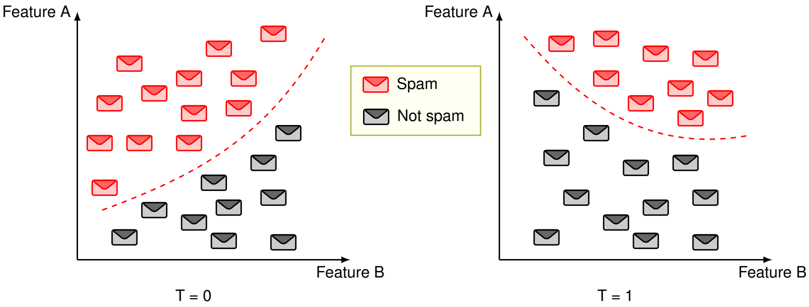


Figure 18.29: Concept drift refers to a change in data patterns and relationships over time.

Distribution shifts like these can dramatically reduce the performance and reliability of ML models in production. Building robust systems requires not only understanding these shifts, but actively detecting and responding to them as they emerge.

18.4.3.2 Mechanisms

Distribution shifts arise from a variety of underlying mechanisms—both natural and system-driven. Understanding these mechanisms helps practitioners detect, diagnose, and design mitigation strategies.

One common mechanism is a change in data sources. When data collected at inference time comes from different sensors, APIs, platforms, or hardware than the training data, even subtle differences in resolution, formatting, or noise can introduce significant shifts. For example, a speech recognition model trained on audio from one microphone type may struggle with data from a different device.

Temporal evolution refers to changes in the underlying data over time. In recommendation systems, user preferences shift. In finance, market conditions change. These shifts may be slow and continuous or abrupt and disruptive. Without temporal awareness or continuous evaluation, models can become obsolete, frequently without prior indication. To illustrate this, Figure 18.30 shows how selective breeding over generations has significantly changed the physical characteristics of a dog breed. The earlier version of the breed exhibits a lean, athletic build, while the modern version is stockier, with a distinctively different head shape and musculature. This transformation is analogous to how data distributions can shift in real-world systems—initial data used to train a model may differ substantially from the data encountered over time. Just as evolutionary pressures shape biological traits, dynamic user behavior, market forces, or changing environments can shift the distribution of data in machine learning applications. Without periodic retraining or adaptation, models exposed to these evolving distributions may underperform or become unreliable.

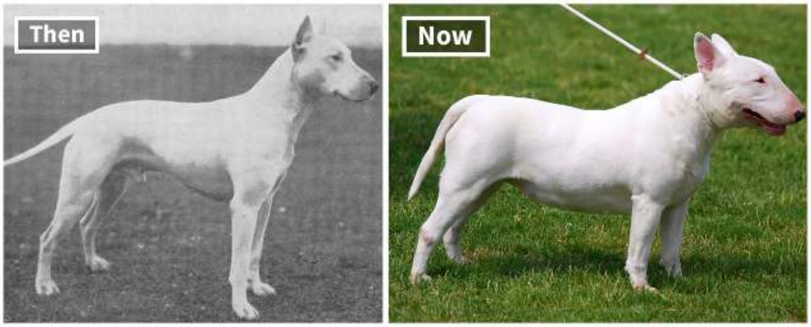


Figure 18.30: Temporal evolution in practice. The evolution of a dog breed over time illustrates how distribution shifts can lead to significant changes in underlying data. In ML systems, similar distribution shifts can occur when models encounter new or evolving data distributions, often requiring retraining or adaptation to maintain accuracy and performance. Without accounting for temporal evolution, models can become misaligned with the target data, leading to degraded performance and unreliable predictions.

Domain-specific variation arises when a model trained on one setting is applied to another. A medical diagnosis model trained on data from one hospital may underperform in another due to differences in equipment, demographics, or clinical workflows. These variations often require explicit adaptation strategies, such as domain generalization or fine-tuning.

Selection bias occurs when the training data does not accurately reflect the target population. This may result from sampling strategies, data access constraints, or labeling choices. The result is a model that overfits to specific segments and fails to generalize. Addressing this requires thoughtful data collection and continuous validation.

Feedback loops are a particularly subtle mechanism. In some systems, model predictions influence user behavior, which in turn affects future inputs. For instance, a dynamic pricing model might set prices that change buying patterns, which then distort the distribution of future training data. These loops can reinforce narrow patterns and make model behavior difficult to predict.

Lastly, adversarial manipulation can induce distribution shifts deliberately. Attackers may introduce out-of-distribution samples or craft inputs that exploit weak spots in the model's decision boundary. These inputs may lie far from the training distribution and can cause unexpected or unsafe predictions.

These mechanisms often interact, making real-world distribution shift detection and mitigation complex. From a systems perspective, this complexity necessitates ongoing monitoring, logging, and feedback pipelines—features often absent in early-stage or static ML deployments.

18.4.3.3 Impact on ML

Distribution shift can affect nearly every dimension of ML system performance, from prediction accuracy and latency to user trust and system maintainability.

A common and immediate consequence is degraded predictive performance. When the data at inference time differs from training data, the model may produce systematically inaccurate or inconsistent predictions. This erosion of accuracy is particularly dangerous in high-stakes applications like fraud detection, autonomous vehicles, or clinical decision support.

Another serious effect is loss of reliability and trustworthiness. As distribution shifts, users may notice inconsistent or erratic behavior. For example, a recommendation system might begin suggesting irrelevant or offensive content. Even if overall accuracy metrics remain acceptable, loss of user trust can undermine the system's value.

Distribution shift also amplifies model bias. If certain groups or data segments are underrepresented in the training data, the model may fail more frequently on those groups. Under shifting conditions, these failures can become more pronounced, resulting in discriminatory outcomes or fairness violations.

There is also a rise in uncertainty and operational risk. In many production settings, model decisions feed directly into business operations or automated actions. Under shift, these decisions become less predictable and harder to validate, increasing the risk of cascading failures or poor decisions downstream.

From a system maintenance perspective, distribution shifts complicate retraining and deployment workflows. Without robust mechanisms for drift detection and performance monitoring, shifts may go unnoticed until performance degrades significantly. Once detected, retraining may be required—raising challenges related to data collection, labeling, model rollback, and validation. This creates friction in continuous integration and deployment (CI/CD) workflows and can significantly slow down iteration cycles.

Moreover, distribution shift increases vulnerability to adversarial attacks. Attackers can exploit the model's poor calibration on unfamiliar data, using slight perturbations to push inputs outside the training distribution and cause failures. This is especially concerning when system feedback loops or automated decisioning pipelines are in place.

From a systems perspective, distribution shift is not just a modeling concern—it is a core operational challenge. It requires end-to-end system support: mechanisms for data logging, drift detection, automated alerts, model versioning, and scheduled retraining. ML systems must be designed to detect when performance degrades in production, diagnose whether a distribution shift is the cause, and trigger appropriate mitigation actions. This might include human-in-the-loop review, fallback strategies, model retraining pipelines, or staged deployment rollouts.

In mature ML systems, handling distribution shift becomes a matter of infrastructure, observability, and automation, not just modeling technique. Failing to account for it risks silent model failure in dynamic, real-world environments—precisely where ML systems are expected to deliver the most value.

A summary of common types of distribution shifts, their effects on model performance, and potential system-level responses is shown in Table 18.3.

18.4.3.4 Summary of Distribution Shifts and System Implications

Table 18.3: Common types of distribution shift, their effects, and system-level mitigations.

Type of Shift	Cause or Example	Consequence for Model	System-Level Response
Covariate Shift	Change in input features (e.g., sensor calibration drift)	Model misclassifies new inputs despite consistent labels	Monitor input distributions; retrain with updated features
Label Shift	Change in label distribution (e.g., new class frequencies in usage)	Prediction probabilities become skewed	Track label priors; reweight or adapt output calibration
Concept Drift	Evolving relationship between inputs and outputs (e.g. fraud tactics)	Model performance degrades over time	Retrain frequently; use continual or online learning
Domain Mismatch	Train on reviews, deploy on tweets	Poor generalization due to different vocabularies or styles	Use domain adaptation or fine-tuning
Contextual Change	New deployment environment (e.g., lighting, user behavior)	Performance varies by context	Collect contextual data; monitor conditional accuracy
Selection Bias	Underrepresentation during training	Biased predictions for unseen groups	Validate dataset balance; augment training data
Feedback Loops	Model outputs affect future inputs (e.g., recommender systems)	Reinforced drift, unpredictable patterns	Monitor feedback effects; consider counterfactual logging
Adversarial Shift	Attackers introduce OOD inputs or perturbations	Model becomes vulnerable to targeted failures	Use robust training; detect out-of-distribution inputs

18.4.4 Detection and Mitigation

The detection and mitigation of threats to ML systems requires combining defensive strategies across multiple layers. These include techniques to identify and counter adversarial attacks, data poisoning attempts, and distribution shifts that can degrade model performance and reliability. Through systematic application of these protections, ML systems can maintain robustness when deployed in dynamic real-world environments.

18.4.4.1 Adversarial Attacks

As discussed earlier, adversarial attacks pose a significant threat to the robustness and reliability of ML systems. These attacks involve crafting carefully designed inputs, known as adversarial examples, to deceive ML models and cause them to make incorrect predictions. To safeguard ML systems against such attacks, it is crucial to develop effective techniques for detecting and mitigating these threats.

Detection Techniques. Detecting adversarial examples is the first line of defense against adversarial attacks. Several techniques have been proposed to identify and flag suspicious inputs that may be adversarial.

Statistical methods aim to detect adversarial examples by analyzing the statistical properties of the input data. These methods often compare the input data distribution to a reference distribution, such as the training data distribution or a known benign distribution. Techniques like the Kolmogorov-Smirnov (Berger and Zhou 2014) test or the Anderson-Darling test can be used to measure the discrepancy between the distributions and flag inputs that deviate significantly from the expected distribution.

Kernel density estimation (KDE) is a non-parametric technique used to estimate the probability density function of a dataset. In the context of adversarial example detection, KDE can be used to estimate the density of benign examples in the input space. Adversarial examples often lie in low-density regions and can be detected by comparing their estimated density to a threshold. Inputs with an estimated density below the threshold are flagged as potential adversarial examples.

Another technique is feature squeezing (Panda, Chakraborty, and Roy 2019), which reduces the complexity of the input space by applying dimensionality reduction or discretization. The idea behind feature squeezing is that adversarial examples often rely on small, imperceptible perturbations that can be eliminated or reduced through these transformations. Inconsistencies can be detected by comparing the model's predictions on the original input and the squeezed input, indicating the presence of adversarial examples.

Model uncertainty estimation techniques try to quantify the confidence or uncertainty associated with a model's predictions. Adversarial examples often exploit regions of high uncertainty in the model's decision boundary. By estimating the uncertainty using techniques like Bayesian neural networks, dropout-based uncertainty estimation, or ensemble methods, inputs with high uncertainty can be flagged as potential adversarial examples.

Defense Strategies. Once adversarial examples are detected, various defense strategies can be employed to mitigate their impact and improve the robustness of ML models.

Adversarial training is a technique that involves augmenting the training data with adversarial examples and retraining the model on this augmented dataset. Exposing the model to adversarial examples during training teaches it to classify them correctly and becomes more robust to adversarial attacks. Adversarial training can be performed using various attack methods, such as the Fast Gradient Sign Method or Projected Gradient Descent (Madry et al. 2017).

Defensive distillation (Papernot et al. 2016) is a technique that trains a second model (the student model) to mimic the behavior of the original model (the teacher model). The student model is trained on the soft labels produced by the teacher model, which are less sensitive to small perturbations. Using the student model for inference can reduce the impact of adversarial perturbations, as the student model learns to generalize better and is less sensitive to adversarial noise.

Input preprocessing and transformation techniques try to remove or mitigate the effect of adversarial perturbations before feeding the input to the ML model. These techniques include image denoising, JPEG compression, random resizing, padding, or applying random transformations to the input data. By reducing the impact of adversarial perturbations, these preprocessing steps can help improve the model's robustness to adversarial attacks.

Ensemble methods combine multiple models to make more robust predictions. The ensemble can reduce the impact of adversarial attacks by using a diverse set of models with different architectures, training data, or hyperparameters. Adversarial examples that fool one model may not fool others in the ensemble, leading to more reliable and robust predictions. Model diversification techniques, such as using different preprocessing techniques or feature representations for each model in the ensemble, can further enhance the robustness.

Evaluation and Testing. Conduct thorough evaluation and testing to assess the effectiveness of adversarial defense techniques and measure the robustness of ML models.

Adversarial robustness metrics quantify the model's resilience to adversarial attacks. These metrics can include the model's accuracy on adversarial examples, the average distortion required to fool the model, or the model's performance under different attack strengths. By comparing these metrics across different models or defense techniques, practitioners can assess and compare their robustness levels.

Standardized adversarial attack benchmarks and datasets provide a common ground for evaluating and comparing the robustness of ML models. These benchmarks include datasets with pre-generated adversarial examples and tools and frameworks for generating adversarial attacks. Examples of popular adversarial attack benchmarks include the MNIST-C, CIFAR-10-C, and ImageNet-C (Hendrycks and Dietterich 2019) datasets, which contain corrupted or perturbed versions of the original datasets.

Practitioners can develop more robust and resilient ML systems by leveraging these adversarial example detection techniques, defense strategies, and robustness evaluation methods. However, it is important to note that adversarial robustness is an ongoing research area, and no single technique provides complete protection against all types of adversarial attacks. A comprehensive approach that combines multiple defense mechanisms and regular testing is essential to maintain the security and reliability of ML systems in the face of evolving adversarial threats.

18.4.4.2 Data Poisoning

Data poisoning attacks aim to corrupt training data used to build ML models, undermining their integrity. As illustrated in Figure 18.31, these attacks can manipulate or pollute the training data in ways that cause models to learn incorrect patterns, leading to erroneous predictions or undesirable behaviors when deployed. Given the foundational role of training data in ML system performance, detecting and mitigating data poisoning is critical for maintaining model trustworthiness and reliability.

Anomaly Detection Techniques. Statistical outlier detection methods identify data points that deviate significantly from most data. These methods assume that poisoned data instances are likely to be statistical outliers. Techniques such as the Z-score method, Tukey's method, or the Mahalanobis distance can be used to measure the deviation of each data point from the central tendency of the dataset. Data points that exceed a predefined threshold are flagged as potential outliers and considered suspicious for data poisoning.

Clustering-based methods group similar data points together based on their features or attributes. The assumption is that poisoned data instances may form

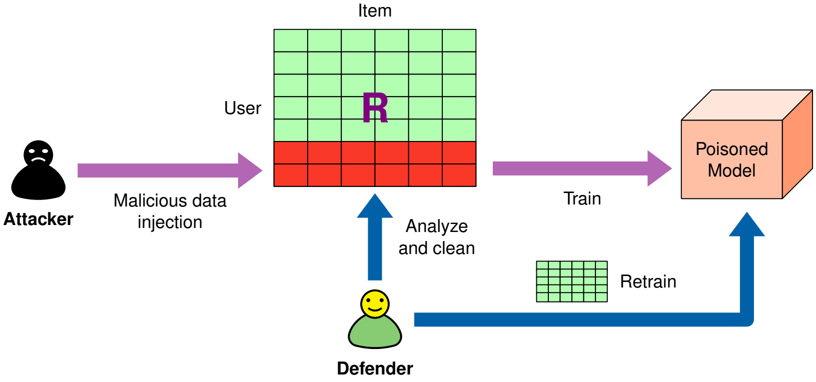


Figure 18.31: Malicious data injection. Source: Li

distinct clusters or lie far away from the normal data clusters. By applying clustering algorithms like K-means, DBSCAN, or hierarchical clustering, anomalous clusters or data points that do not belong to any cluster can be identified. These anomalous instances are then treated as potentially poisoned data.

Autoencoders are neural networks trained to reconstruct the input data from a compressed representation, as shown in Figure 18.32. They can be used for anomaly detection by learning the normal patterns in the data and identifying instances that deviate from them. During training, the autoencoder is trained on clean, unpoisoned data. At inference time, the reconstruction error for each data point is computed. Data points with high reconstruction errors are considered abnormal and potentially poisoned, as they do not conform to the learned normal patterns.

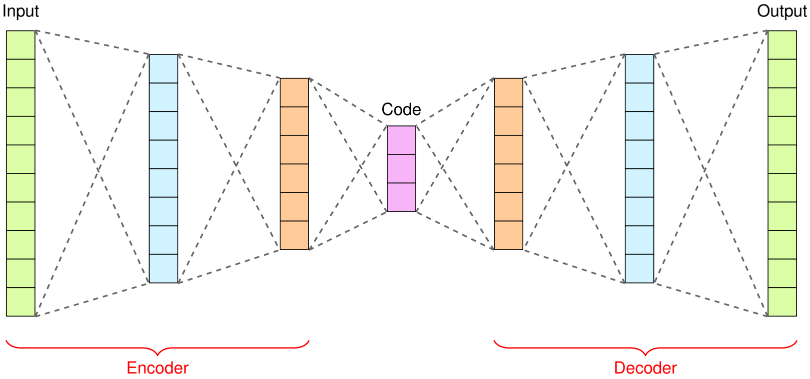


Figure 18.32: Autoencoder. Source: Dertat

Sanitization and Preprocessing. Data poisoning can be avoided by cleaning data, which involves identifying and removing or correcting noisy, incomplete, or inconsistent data points. Techniques such as data deduplication, missing value imputation, and outlier removal can be applied to improve the quality of the training data. By eliminating or filtering out suspicious or anomalous data points, the impact of poisoned instances can be reduced.

Data validation involves verifying the integrity and consistency of the training data. This can include checking for data type consistency, range validation, and cross-field dependencies. By defining and enforcing data validation rules, anomalous or inconsistent data points indicative of data poisoning can be identified and flagged for further investigation.

Data provenance and lineage tracking involve maintaining a record of data's origin, transformations, and movements throughout the ML pipeline. By documenting the data sources, preprocessing steps, and any modifications made to the data, practitioners can trace anomalies or suspicious patterns back to their origin. This helps identify potential points of data poisoning and facilitates the investigation and mitigation process.

Robust Training. Robust optimization techniques can be used to modify the training objective to minimize the impact of outliers or poisoned instances. This can be achieved by using robust loss functions less sensitive to extreme values, such as the Huber loss or the modified Huber loss[18]. Regularization techniques[19], such as L1 or L2 regularization, can also help in reducing the model's sensitivity to poisoned data by constraining the model's complexity and preventing overfitting.

Robust loss functions are designed to be less sensitive to outliers or noisy data points. Examples include the modified Huber loss, the Tukey loss (Beaton and Tukey 1974), and the trimmed mean loss. These loss functions down-weight or ignore the contribution of abnormal instances during training, reducing their impact on the model's learning process. Robust objective functions, such as the minimax[20] or distributionally robust objective, aim to optimize the model's performance under worst-case scenarios or in the presence of adversarial perturbations.

Data augmentation techniques involve generating additional training examples by applying random transformations or perturbations to the existing data Figure 18.33. This helps in increasing the diversity and robustness of the training dataset. By introducing controlled variations in the data, the model becomes less sensitive to specific patterns or artifacts that may be present in poisoned instances. Randomization techniques, such as random subsampling or bootstrap aggregating, can also help reduce the impact of poisoned data by training multiple models on different subsets of the data and combining their predictions.

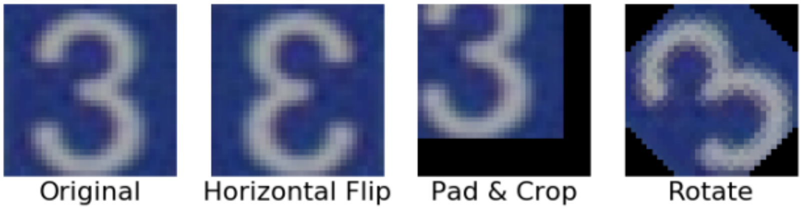


Figure 18.33: An image of the number "3" in original form and with basic augmentations applied.

Secure Data Sourcing. Implementing the best data collection and curation practices can help mitigate the risk of data poisoning. This includes establishing clear data collection protocols, verifying the authenticity and reliability of

[18] Huber Loss: A loss function used in robust regression that is less sensitive to outliers in data than squared error loss.

[19] Regularization: A method used in neural networks to prevent overfitting in models by adding a cost term to the loss function.

[20] Minimax: A decision-making strategy, used in game theory and decision theory, which tries to minimize the maximum possible loss.

data sources, and conducting regular data quality assessments. Sourcing data from trusted and reputable providers and following secure data handling practices can reduce the likelihood of introducing poisoned data into the training pipeline.

Strong data governance and access control mechanisms are essential to prevent unauthorized modifications or tampering with the training data. This involves defining clear roles and responsibilities for data access, implementing access control policies based on the principle of least privilege,[21] and monitoring and logging data access activities. By restricting access to the training data and maintaining an audit trail, potential data poisoning attempts can be detected and investigated.

Detecting and mitigating data poisoning attacks requires a multifaceted approach that combines anomaly detection, data sanitization,[22] robust training techniques, and secure data sourcing practices. By implementing these measures, ML practitioners can improve the resilience of their models against data poisoning and ensure the integrity and trustworthiness of the training data. However, it is important to note that data poisoning is an active area of research, and new attack vectors and defense mechanisms continue to emerge. Staying informed about the latest developments and adopting a proactive and adaptive approach to data security is crucial for maintaining the robustness of ML systems.

18.4.4.3 Distribution Shifts

Detection and Mitigation. Recall that distribution shifts occur when the data distribution encountered by a machine learning (ML) model during deployment differs from the distribution it was trained on. These shifts can significantly impact the model's performance and generalization ability, leading to suboptimal or incorrect predictions. Detecting and mitigating distribution shifts is crucial to ensure the robustness and reliability of ML systems in real-world scenarios.

Detection Techniques. Statistical tests can be used to compare the distributions of the training and test data to identify significant differences. Techniques such as the Kolmogorov-Smirnov test or the Anderson-Darling test measure the discrepancy between two distributions and provide a quantitative assessment of the presence of distribution shift. By applying these tests to the input features or the model's predictions, practitioners can detect if there is a statistically significant difference between the training and test distributions.

Divergence metrics quantify the dissimilarity between two probability distributions. Commonly used divergence metrics include the Kullback-Leibler (KL) divergence and the Jensen-Shannon (JS) divergence. By calculating the divergence between the training and test data distributions, practitioners can assess the extent of the distribution shift. High divergence values indicate a significant difference between the distributions, suggesting the presence of a distribution shift.

Uncertainty quantification techniques, such as Bayesian neural networks[23] or ensemble methods[24], can estimate the uncertainty associated with the model's predictions. When a model is applied to data from a different distribution, its predictions may have higher uncertainty. By monitoring the uncertainty

[21] Principle of Least Privilege: A security concept in which a user is given the minimum levels of access necessary to complete his/her job functions.

[22] Data Sanitization: The process of deliberately, permanently, and irreversibly removing or destroying the data stored on a memory device to make it unrecoverable.

[23] Bayesian Neural Networks: Neural networks that incorporate probability distributions over their weights, enabling uncertainty quantification in predictions and more robust decision making.

[24] Ensemble Methods: An ML approach that combines several models to improve prediction accuracy.

levels, practitioners can detect distribution shifts. If the uncertainty consistently exceeds a predetermined threshold for test samples, it suggests that the model is operating outside its trained distribution.

In addition, domain classifiers are trained to distinguish between different domains or distributions. Practitioners can detect distribution shifts by training a classifier to differentiate between the training and test domains. If the domain classifier achieves high accuracy in distinguishing between the two domains, it indicates a significant difference in the underlying distributions. The performance of the domain classifier serves as a measure of the distribution shift.

Mitigation Techniques. Transfer learning leverages knowledge gained from one domain to improve performance in another, as shown in Figure 18.34. By using pre-trained models or transferring learned features from a source domain to a target domain, transfer learning can help mitigate the impact of distribution shifts. The pre-trained model can be fine-tuned on a small amount of labeled data from the target domain, allowing it to adapt to the new distribution. Transfer learning is particularly effective when the source and target domains share similar characteristics or when labeled data in the target domain is scarce.

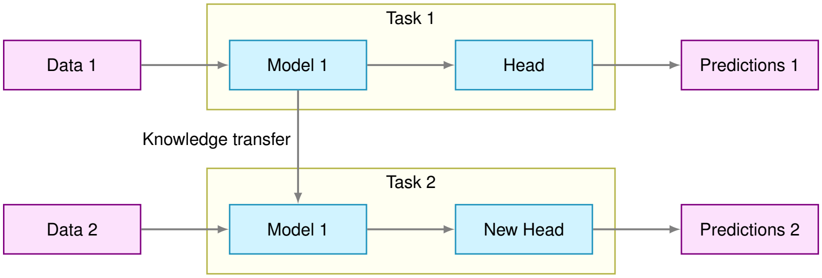


Figure 18.34: Transfer learning. Source: Bhavsar

Continual learning, also known as lifelong learning, enables ML models to learn continuously from new data distributions while retaining knowledge from previous distributions. Techniques such as elastic weight consolidation (EWC) (Kirkpatrick et al. 2017) or gradient episodic memory (GEM) (Lopez-Paz and Ranzato 2017) allow models to adapt to evolving data distributions over time. These techniques aim to balance the plasticity of the model (ability to learn from new data) with the stability of the model (retaining previously learned knowledge). By incrementally updating the model with new data and mitigating catastrophic forgetting, continual learning helps models stay robust to distribution shifts.

Data augmentation techniques, such as those we have seen previously, involve applying transformations or perturbations to the existing training data to increase its diversity and improve the model's robustness to distribution shifts. By introducing variations in the data, such as rotations, translations, scaling, or adding noise, data augmentation helps the model learn invariant features and generalize better to unseen distributions. Data augmentation can be performed during training and inference to improve the model's ability to handle distribution shifts.

Ensemble methods combine multiple models to make predictions more robust to distribution shifts. By training models on different subsets of the data, using different algorithms, or with different hyperparameters, ensemble methods can capture diverse aspects of the data distribution. When presented with a shifted distribution, the ensemble can leverage the strengths of individual models to make more accurate and stable predictions. Techniques like bagging, boosting, or stacking can create effective ensembles.

Regularly updating models with new data from the target distribution is crucial to mitigate the impact of distribution shifts. As the data distribution evolves, models should be retrained or fine-tuned on the latest available data to adapt to the changing patterns. Monitoring model performance and data characteristics can help detect when an update is necessary. By keeping the models up to date, practitioners can ensure they remain relevant and accurate in the face of distribution shifts.

Evaluating models using robust metrics less sensitive to distribution shifts can provide a more reliable assessment of model performance. Metrics such as the area under the precision-recall curve (AUPRC) or the F1 score[25] are more robust to class imbalance and can better capture the model's performance across different distributions. Additionally, using domain-specific evaluation metrics that align with the desired outcomes in the target domain can provide a more meaningful measure of the model's effectiveness.

Detecting and mitigating distribution shifts is an ongoing process that requires continuous monitoring, adaptation, and improvement. By employing a combination of detection techniques and mitigation strategies, ML practitioners can proactively identify and address distribution shifts, ensuring the robustness and reliability of their models in real-world deployments. It is important to note that distribution shifts can take various forms and may require domain-specific approaches depending on the nature of the data and the application. Staying informed about the latest research and best practices in handling distribution shifts is essential for building resilient ML systems.

18.5 Software Faults

Machine learning systems rely on complex software infrastructures that extend far beyond the models themselves. These systems are built on top of frameworks, libraries, and runtime environments that facilitate model training, evaluation, and deployment. As with any large-scale software system, the components that support ML workflows are susceptible to faults—unintended behaviors resulting from defects, bugs, or design oversights in the software. These faults can manifest across all stages of an ML pipeline and, if not identified and addressed, may impair performance, compromise security, or even invalidate results. This section examines the nature, causes, and consequences of software faults in ML systems, as well as strategies for their detection and mitigation.

18.5.1 Characteristics

Software faults in ML frameworks originate from various sources, including programming errors, architectural misalignments, and version incompatibili-

[25] F1 Score: A measure of a model's accuracy that combines precision (correct positive predictions) and recall (proportion of actual positives identified) into a single metric. Calculated as the harmonic mean of precision and recall.

ties. These faults exhibit several important characteristics that influence how they arise and propagate in practice.

One defining feature of software faults is their diversity. Faults can range from syntactic and logical errors to more complex manifestations such as memory leaks, concurrency bugs, or failures in integration logic. The broad variety of potential fault types complicates both their identification and resolution, as they often surface in non-obvious ways.

A second key characteristic is their tendency to propagate across system boundaries. An error introduced in a low-level module, such as a tensor allocation routine or a preprocessing function, can produce cascading effects that disrupt model training, inference, or evaluation. Because ML frameworks are often composed of interconnected components, a fault in one part of the pipeline can introduce failures in seemingly unrelated modules.

Some faults are intermittent, manifesting only under specific conditions such as high system load, particular hardware configurations, or rare data inputs. These transient faults are notoriously difficult to reproduce and diagnose, as they may not consistently appear during standard testing procedures.

Furthermore, software faults may subtly interact with ML models themselves. For example, a bug in a data transformation script might introduce systematic noise or shift the distribution of inputs, leading to biased or inaccurate predictions. Similarly, faults in the serving infrastructure may result in discrepancies between training-time and inference-time behaviors, undermining deployment consistency.

The consequences of software faults extend to a range of system properties. Faults may impair performance by introducing latency or inefficient memory usage; they may reduce scalability by limiting parallelism; or they may compromise reliability and security by exposing the system to unexpected behaviors or malicious exploitation.

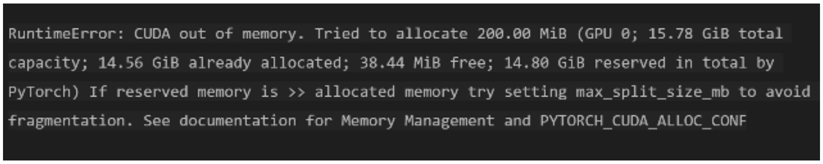
Finally, the manifestation of software faults is often shaped by external dependencies, such as hardware platforms, operating systems, or third-party libraries. Incompatibilities arising from version mismatches or hardware-specific behavior may result in subtle, hard-to-trace bugs that only appear under certain runtime conditions.

A thorough understanding of these characteristics is essential for developing robust software engineering practices in ML. It also provides the foundation for the detection and mitigation strategies described later in this section.

18.5.2 Mechanisms

Software faults in ML frameworks arise through a variety of mechanisms, reflecting the complexity of modern ML pipelines and the layered architecture of supporting tools. These mechanisms correspond to specific classes of software failures that commonly occur in practice.

One prominent class involves resource mismanagement, particularly with respect to memory. Improper memory allocation, including the failure to release buffers or file handles, can lead to memory leaks and, eventually, to resource exhaustion. This is especially detrimental in deep learning applications, where large tensors and GPU memory allocations are common. As shown in Figure 18.35, inefficient memory usage or the failure to release GPU resources can cause training procedures to halt or significantly degrade runtime performance.



```
RuntimeError: CUDA out of memory. Tried to allocate 200.00 MiB (GPU 0; 15.78 GiB total
capacity; 14.56 GiB already allocated; 38.44 MiB free; 14.80 GiB reserved in total by
PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb to avoid
fragmentation. See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

Figure 18.35: Example of GPU out-of-the-memory and suboptimal utilization issues

Another recurring fault mechanism stems from concurrency and synchronization errors. In distributed or multi-threaded environments, incorrect coordination among parallel processes can lead to race conditions, deadlocks, or inconsistent states. These issues are often tied to the improper use of asynchronous operations, such as non-blocking I/O or parallel data ingestion. Synchronization bugs can corrupt the consistency of training states or produce unreliable model checkpoints.

Compatibility problems frequently arise from changes to the software environment. For example, upgrading a third-party library without validating downstream effects may introduce subtle behavioral changes or break existing functionality. These issues are exacerbated when the training and inference environments differ in hardware, operating system, or dependency versions. Reproducibility in ML experiments often hinges on managing these environmental inconsistencies.

Faults related to numerical instability are also common in ML systems, particularly in optimization routines. Improper handling of floating-point precision, division by zero, or underflow/overflow conditions can introduce instability into gradient computations and convergence procedures. As described in this resource, the accumulation of rounding errors across many layers of computation can distort learned parameters or delay convergence.

Exception handling, though often overlooked, plays a crucial role in the stability of ML pipelines. Inadequate or overly generic exception management can cause systems to fail silently or crash under non-critical errors. Moreover, ambiguous error messages and poor logging practices impede diagnosis and prolong resolution times.

These fault mechanisms, while diverse in origin, share the potential to significantly impair ML systems. Understanding how they arise provides the basis for effective system-level safeguards.

18.5.3 Impact on ML

The consequences of software faults can be profound, affecting not only the correctness of model outputs but also the broader usability and reliability of an ML system in production.

Performance degradation is a common symptom, often resulting from memory leaks, inefficient resource scheduling, or contention between concurrent threads. These issues tend to accumulate over time, leading to increased latency, reduced throughput, or even system crashes. As noted by (Maas et al. 2024), the accumulation of performance regressions across components can severely restrict the operational capacity of ML systems deployed at scale.

In addition to slowing system performance, faults can lead to inaccurate predictions. For example, preprocessing errors or inconsistencies in feature encoding can subtly alter the input distribution seen by the model, producing biased or unreliable outputs. These kinds of faults are particularly insidious, as they may not trigger any obvious failure but still compromise downstream decisions. Over time, rounding errors and precision loss can amplify inaccuracies, particularly in deep architectures with many layers or long training durations.

Reliability is also undermined by software faults. Systems may crash unexpectedly, fail to recover from errors, or behave inconsistently across repeated executions. Intermittent faults are especially problematic in this context, as they erode user trust while eluding conventional debugging efforts. In distributed settings, faults in checkpointing or model serialization can cause training interruptions or data loss, reducing the resilience of long-running training pipelines.

Security vulnerabilities frequently arise from overlooked software faults. Buffer overflows, improper validation, or unguarded inputs can open the system to manipulation or unauthorized access. Attackers may exploit these weaknesses to alter the behavior of models, extract private data, or induce denial-of-service conditions. As described by (Q. Li et al. 2023), such vulnerabilities pose serious risks, particularly when ML systems are integrated into critical infrastructure or handle sensitive user data.

Moreover, the presence of faults complicates development and maintenance. Debugging becomes more time-consuming, especially when fault behavior is non-deterministic or dependent on external configurations. Frequent software updates or library patches may introduce regressions that require repeated testing. This increased engineering overhead can slow iteration, inhibit experimentation, and divert resources from model development.

Taken together, these impacts underscore the importance of systematic software engineering practices in ML—practices that anticipate, detect, and mitigate the diverse failure modes introduced by software faults.

18.5.4 Detection and Mitigation

Addressing software faults in ML systems requires an integrated strategy that spans development, testing, deployment, and monitoring. A comprehensive mitigation framework should combine proactive detection methods with robust design patterns and operational safeguards.

To help summarize these techniques and clarify where each strategy fits in the ML lifecycle, Table 18.4 below categorizes detection and mitigation approaches by phase and objective. This table provides a high-level overview that complements the detailed explanations that follow.

Table 18.4: Summary of detection and mitigation techniques for software faults.

Category	Technique	Purpose	When to Apply
Testing and Validation	Unit testing, integration testing, regression testing	Verify correctness and identify regressions	During development

Category	Technique	Purpose	When to Apply
Static Analysis and Linting	Static analyzers, linters, code reviews	Detect syntax errors, unsafe operations, enforce best practices	Before integration
Runtime Monitoring & Logging	Metric collection, error logging, profiling | Observe system behavior, detect anomalies | During training and deployment |		
Fault-Tolerant Design	Exception handling, modular architecture, checkpointing	Minimize impact of failures, support recovery	Design and implementation phase
Update Management	Dependency auditing, test staging, version tracking	Prevent regressions and compatibility issues	Before system upgrades or deployment
Environment Isolation	Containerization (e.g., Docker, Kubernetes), virtual environments	Ensure reproducibility, avoid environment-specific bugs	Development, testing, deployment
CI/CD and Automation	Automated test pipelines, monitoring hooks, deployment gates	Enforce quality assurance and catch faults early	Continuously throughout development

The first line of defense involves systematic testing. Unit testing verifies that individual components behave as expected under normal and edge-case conditions. Integration testing ensures that modules interact correctly across boundaries, while regression testing detects errors introduced by code changes. Continuous testing is essential in fast-moving ML environments, where pipelines evolve rapidly and small modifications may have system-wide consequences. As shown in Figure 18.36, automated regression tests help preserve functional correctness over time.

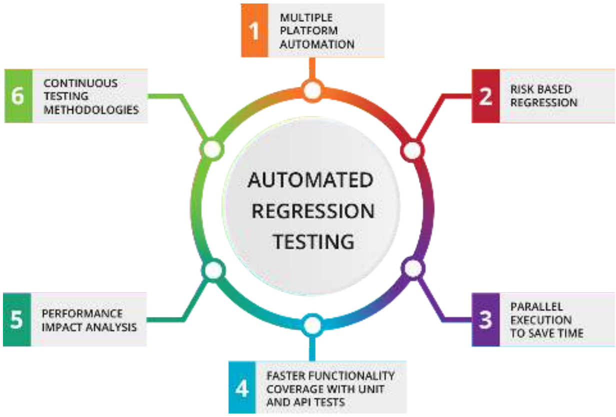

1 MULTIPLE PLATFORM AUTOMATION
6 CONTINUOUS TESTING METHODOLOGIES
2 RISK BASED REGRESSION
AUTOMATED REGRESSION TESTING
5 PERFORMANCE IMPACT ANALYSIS
3 PARALLEL EXECUTION TO SAVE TIME
4 FASTER FUNCTIONALITY COVERAGE WITH UNIT AND API TESTS

Figure 18.36: Automated regression testing. Source: UTOR

Static code analysis tools complement dynamic tests by identifying potential issues at compile time. These tools catch common errors such as variable misuse, unsafe operations, or violation of language-specific best practices. Combined with code reviews and consistent style enforcement, static analysis reduces the incidence of avoidable programming faults.

Runtime monitoring is critical for observing system behavior under real-world conditions. Logging frameworks should capture key signals such as memory usage, input/output traces, and exception events. Monitoring tools can track model throughput, latency, and failure rates, providing early warnings of software faults. Profiling, as illustrated in this Microsoft resource, helps identify performance bottlenecks and inefficiencies indicative of deeper architectural issues.

Robust system design further improves fault tolerance. Structured exception handling and assertion checks prevent small errors from cascading into system-wide failures. Redundant computations, fallback models, and failover mechanisms improve availability in the presence of component failures. Modular architectures that encapsulate state and isolate side effects make it easier to diagnose and contain faults. Checkpointing techniques, such as those discussed in (Eisenman et al. 2022), enable recovery from mid-training interruptions without data loss.

Keeping ML software up to date is another key strategy. Applying regular updates and security patches helps address known bugs and vulnerabilities. However, updates must be validated through test staging environments to avoid regressions. Reviewing release notes and change logs ensures teams are aware of any behavioral changes introduced in new versions.

Containerization technologies like Docker and Kubernetes allow teams to define reproducible runtime environments that mitigate compatibility issues. By isolating system dependencies, containers prevent faults introduced by system-level discrepancies across development, testing, and production.

Finally, automated pipelines built around continuous integration and continuous deployment (CI/CD) provide an infrastructure for enforcing fault-aware development. Testing, validation, and monitoring can be embedded directly into the CI/CD flow. As shown in Figure 18.37, such pipelines reduce the risk of unnoticed regressions and ensure only tested code reaches deployment environments.

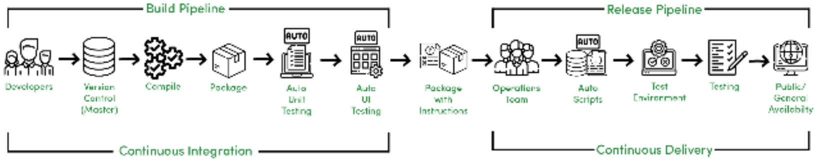


Figure 18.37: Continuous Integration/Continuous Deployment (CI/CD) procedure. Source: geeksforgeeks

Together, these practices form a holistic approach to software fault management in ML systems. When adopted comprehensively, they reduce the likelihood of system failures, improve long-term maintainability, and foster trust in model performance and reproducibility.

18.6 Tools and Frameworks

Given the importance of developing robust AI systems, in recent years, researchers and practitioners have developed a wide range of tools and frameworks to understand how hardware faults manifest and propagate to impact ML systems. These tools and frameworks play a crucial role in evaluating the resilience of ML systems to hardware faults by simulating various fault scenarios and analyzing their impact on the system's performance. This enables designers to identify potential vulnerabilities and develop effective mitigation strategies, ultimately creating more robust and reliable ML systems that can operate safely despite hardware faults. This section provides an overview of widely used fault models in the literature and the tools and frameworks developed to evaluate the impact of such faults on ML systems.

18.6.1 Fault and Error Models

As discussed previously, hardware faults can manifest in various ways, including transient, permanent, and intermittent faults. In addition to the type of fault under study, how the fault manifests is also important. For example, does the fault happen in a memory cell or during the computation of a functional unit? Is the impact on a single bit, or does it impact multiple bits? Does the fault propagate all the way and impact the application (causing an error), or does it get masked quickly and is considered benign? All these details impact what is known as the fault model, which plays a major role in simulating and measuring what happens to a system when a fault occurs.

To effectively study and understand the impact of hardware faults on ML systems, it is essential to understand the concepts of fault models and error models. A fault model describes how a hardware fault manifests itself in the system, while an error model represents how the fault propagates and affects the system's behavior.

Fault models are often classified by several key properties. First, they can be defined by their duration: transient faults are temporary and vanish quickly; permanent faults persist indefinitely; and intermittent faults occur sporadically, making them particularly difficult to identify or predict. Another dimension is fault location, with faults arising in hardware components such as memory cells, functional units, or interconnects. Faults can also be characterized by their granularity—some faults affect only a single bit (e.g., a bitflip), while others impact multiple bits simultaneously, as in burst errors.

Error models, in contrast, describe the behavioral effects of faults as they propagate through the system. These models help researchers understand how initial hardware-level disturbances might manifest in the system's behavior, such as through corrupted weights or miscomputed activations in an ML model. These models may operate at various abstraction levels, from low-level hardware errors to higher-level logical errors in ML frameworks.

The choice of fault or error model is central to robustness evaluation. For example, a system built to study single-bit transient faults (Sangchoolie, Pattabiraman, and Karlsson 2017) will not offer meaningful insight into the effects of permanent multi-bit faults (Wilkening et al. 2014), since its design and assumptions are grounded in a different fault model entirely.

It's also important to consider how and where an error model is implemented. A single-bit flip at the architectural register level, modeled using simulators like gem5 (Binkert et al. 2011), differs meaningfully from a similar bit flip in a PyTorch model's weight tensor. While both simulate value-level perturbations, the lower-level model captures microarchitectural effects that are often abstracted away in software frameworks.

Interestingly, certain fault behavior patterns remain consistent regardless of abstraction level. For example, research has consistently demonstrated that single-bit faults cause more disruption than multi-bit faults, whether examining hardware-level effects or software-visible impacts (Sangchoolie, Pattabiraman, and Karlsson 2017; Papadimitriou and Gizopoulos 2021). However, other important behaviors like error masking (Mohanram and Touba, n.d.) may only be observable at lower abstraction levels. As illustrated in Figure 18.38, this masking phenomenon can cause faults to be filtered out before they propagate to higher levels, meaning software-based tools may miss these effects entirely.

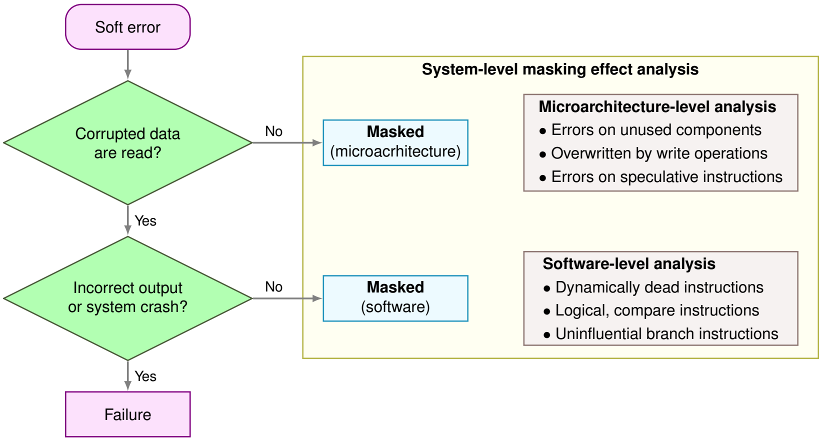

Soft error
Corrupted data are read?
No
Yes
Incorrect output or system crash?
No
Yes
Failure
System-level masking effect analysis
Masked (microacrhitecture)
Masked (software)
Microarchitecture-level analysis
• Errors on unused components
• Overwritten by write operations
• Errors on speculative instructions
Software-level analysis
• Dynamically dead instructions
• Logical, compare instructions
• Uninfluential branch instructions

Figure 18.38: Example of error masking in microarchitectural components (Ko 2021)

To address these discrepancies, tools like Fidelity (Yi He, Balaprakash, and Li 2020) have been developed to align fault models across abstraction layers. By mapping software-observed fault behaviors to corresponding hardware-level patterns (E. Cheng et al. 2016), Fidelity offers a more accurate means of simulating hardware faults at the software level. While lower-level tools capture the true propagation of errors through a hardware system, they are generally slower and more complex. Software-level tools, such as those implemented in PyTorch or TensorFlow, are faster and easier to use for large-scale robustness testing, albeit with less precision.

18.6.2 Hardware-Based Fault Injection

Hardware-based fault injection methods allow researchers to directly introduce faults into physical systems and observe their effects on machine learning (ML) models. These approaches are essential for validating assumptions made in software-level fault injection tools and for studying how real-world hardware faults influence system behavior. While most error injection tools used in ML robustness research are software-based, because of their speed and scalability, hardware-based approaches remain critical for grounding higher-level error models. They are considered the most accurate means of studying the impact of faults on ML systems by manipulating the hardware directly to introduce errors.

As illustrated in Figure 18.39, hardware faults can arise at various points within a deep neural network (DNN) processing pipeline. These faults may affect the control unit, on-chip memory (SRAM), off-chip memory (DRAM), processing elements, and accumulators, leading to erroneous results. In the depicted example, a DNN tasked with recognizing traffic signals correctly identifies a red light under normal conditions. However, hardware-induced faults, caused by phenomena such as aging, electromigration, soft errors, process variations, and manufacturing defects, can introduce errors that cause the DNN to misclassify the signal as a green light, potentially leading to catastrophic consequences in real-world applications.

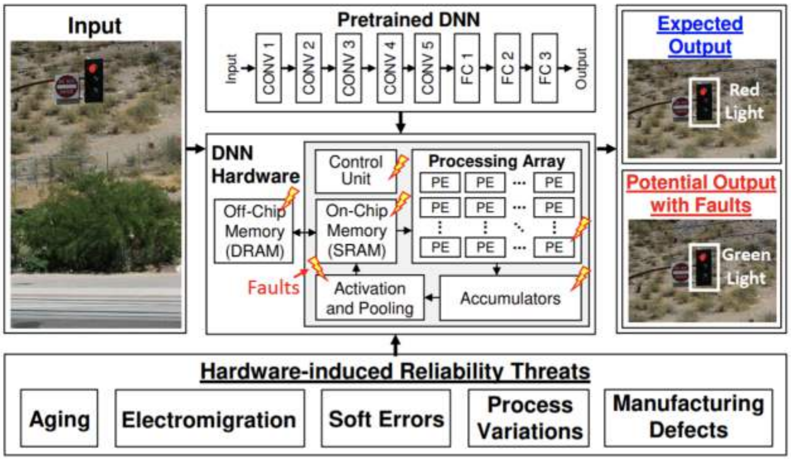


Figure 18.39: Hardware errors can occur due to a variety of reasons and at different times and/or locations in a system, which can be explored when studying the impact of hardware-based errors on systems (Ahmadilivani et al. 2024)

These methods enable researchers to observe the system's behavior under real-world fault conditions. Both software-based and hardware-based error injection tools are described in this section in more detail.

18.6.2.1 Methods

Two of the most common hardware-based fault injection methods are FPGA-based fault injection and radiation or beam testing.

FPGA-based Fault Injection. Field-Programmable Gate Arrays (FPGAs) are reconfigurable integrated circuits that can be programmed to implement various hardware designs. In the context of fault injection, FPGAs offer high precision and accuracy, as researchers can target specific bits or sets of bits within the hardware. By modifying the FPGA configuration, faults can be introduced at specific locations and times during the execution of an ML model. FPGA-based fault injection allows for fine-grained control over the fault model, enabling researchers to study the impact of different types of faults, such as single-bit flips or multi-bit errors. This level of control makes FPGA-based fault injection a valuable tool for understanding the resilience of ML systems to hardware faults.

While FPGA-based methods allow precise, controlled fault injection, other approaches aim to replicate fault conditions found in natural environments.

Radiation or Beam Testing. Radiation or beam testing (Velazco, Foucard, and Peronnard 2010) exposes hardware running ML models to high-energy particles like protons or neutrons. As shown in Figure 18.40, specialized test facilities enable controlled radiation exposure to induce bitflips and other hardware-level faults. This approach is widely regarded as one of the most accurate methods for measuring error rates from particle strikes during application execution. Beam testing provides highly realistic fault scenarios that mirror conditions in radiation-rich environments, making it particularly valuable for validating systems destined for space missions or particle physics experiments. However, while beam testing offers exceptional realism, it lacks the precise targeting capabilities of FPGA-based injection - particle beams cannot be aimed at specific hardware bits or components with high precision. Despite this limitation and its significant operational complexity and cost, beam testing remains a trusted industry practice for rigorously evaluating hardware reliability under real-world radiation effects.

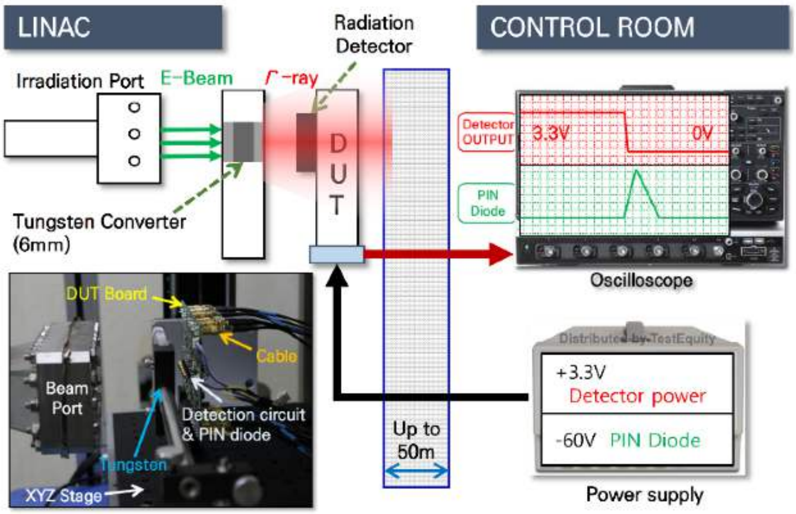


Figure 18.40: Radiation test setup for semiconductor components (Lee et al. 2022) Source: JD Instrument

18.6.2.2 Limitations

Despite their high accuracy, hardware-based fault injection methods have several limitations that can hinder their widespread adoption.

First, cost is a major barrier. Both FPGA-based and beam testing[26] approaches require specialized hardware and facilities, which can be expensive to set up and maintain. This makes them less accessible to research groups with limited funding or infrastructure.

Second, these methods face challenges in scalability. Injecting faults and collecting data directly on hardware is time-consuming, which limits the number of experiments that can be run in a reasonable timeframe. This is especially restrictive when analyzing large ML systems or performing statistical evaluations across many fault scenarios.

[26] Beam Testing: A testing method that exposes hardware to controlled particle radiation to evaluate its resilience to soft errors. Common in aerospace, medical devices, and high-reliability computing.

Third, there are flexibility limitations. Hardware-based methods may not be as adaptable as software-based alternatives when modeling a wide variety of fault and error types. Changing the experimental setup to accommodate a new fault model often requires time-intensive hardware reconfiguration.

Despite these limitations, hardware-based fault injection remains essential for validating the accuracy of software-based tools and for studying system behavior under real-world fault conditions. By combining the high fidelity of hardware-based methods with the scalability and flexibility of software-based tools, researchers can develop a more complete understanding of ML systems' resilience to hardware faults and craft effective mitigation strategies.

18.6.3 Software-Based Fault Injection

As machine learning frameworks like TensorFlow, PyTorch, and Keras have become the dominant platforms for developing and deploying ML models, software-based fault injection tools have emerged as a flexible and scalable way to evaluate the robustness of these systems to hardware faults. Unlike hardware-based approaches, which operate directly on physical systems, software-based methods simulate the effects of hardware faults by modifying a model's underlying computational graph, tensor values, or intermediate computations.

These tools have become increasingly popular in recent years because they integrate directly with ML development pipelines, require no specialized hardware, and allow researchers to conduct large-scale fault injection experiments quickly and cost-effectively. By simulating hardware-level faults, including bit flips in weights, activations, or gradients, at the software level, these tools enable efficient testing of fault tolerance mechanisms and provide valuable insight into model vulnerabilities.

In the remainder of this section, we will examine the advantages and limitations of software-based fault injection methods, introduce major classes of tools (both general-purpose and domain-specific), and discuss how they contribute to building resilient ML systems.

18.6.3.1 Advantages and Trade-offs

Software-based fault injection tools offer several advantages that make them attractive for studying the resilience of ML systems.

One of the primary benefits is speed. Since these tools operate entirely within the software stack, they avoid the overhead associated with modifying physical hardware or configuring specialized test environments. This efficiency enables researchers to perform a large number of fault injection experiments in significantly less time. The ability to simulate a wide range of faults quickly makes these tools particularly useful for stress-testing large-scale ML models or conducting statistical analyses that require thousands of injections.

Another major advantage is flexibility. Software-based fault injectors can be easily adapted to model various types of faults. Researchers can simulate single-bit flips, multi-bit corruptions, or even more complex behaviors such as burst errors or partial tensor corruption. Additionally, software tools allow faults to be injected at different stages of the ML pipeline, at the stages of training, inference, or gradient computation, enabling precise targeting of different system components or layers.

These tools are also highly accessible, as they require only standard ML development environments. Unlike hardware-based methods, there is no need for costly experimental setups, custom circuitry, or radiation testing facilities. This accessibility opens up fault injection research to a broader range of institutions and developers, including those working in academia, startups, or resource-constrained environments.

However, these advantages come with certain trade-offs. Chief among them is accuracy. Because software-based tools model faults at a higher level of abstraction, they may not fully capture the low-level hardware interactions that influence how faults actually propagate. For example, a simulated bit flip in an ML framework may not account for how data is buffered, cached, or manipulated at the hardware level, potentially leading to oversimplified conclusions.

Closely related is the issue of fidelity. While it is possible to approximate real-world fault behaviors, software-based tools may diverge from true hardware behavior, particularly when it comes to subtle interactions like masking, timing, or data movement. The results of such simulations depend heavily on the underlying assumptions of the error model and may require validation against real hardware measurements to be reliable.

Despite these limitations, software-based fault injection tools play an indispensable role in the study of ML robustness. Their speed, flexibility, and accessibility allow researchers to perform wide-ranging evaluations and inform the development of fault-tolerant ML architectures. In subsequent sections, we explore the major tools in this space, highlighting their capabilities and use cases.

18.6.3.2 Limitations

While software-based fault injection tools offer significant advantages in terms of speed, flexibility, and accessibility, they are not without limitations. These constraints can impact the accuracy and realism of fault injection experiments, particularly when assessing the robustness of ML systems to real-world hardware faults.

One major concern is accuracy. Because software-based tools operate at higher levels of abstraction, they may not always capture the full spectrum of effects that hardware faults can produce. Low-level hardware interactions, including subtle timing errors, voltage fluctuations, and architectural side effects, can be missed entirely in high-level simulations. As a result, fault injection studies that rely solely on software models may under- or overestimate a system's true vulnerability to certain classes of faults.

Closely related is the issue of fidelity. While software-based methods are often designed to emulate specific fault behaviors, the extent to which they reflect real-world hardware conditions can vary. For example, simulating a single-bit flip in the value of a neural network weight may not fully replicate how that same bit error would propagate through memory hierarchies or affect computation units on an actual chip. The more abstract the tool, the greater the risk that the simulated behavior will diverge from physical behavior under fault conditions.

Moreover, because software-based tools are easier to modify, there is a risk of unintentionally deviating from realistic fault assumptions. This can occur if the chosen fault model is overly simplified or not grounded in empirical data from actual hardware behavior. As discussed later in the section on bridging the hardware-software gap, tools like Fidelity (Yi He, Balaprakash, and Li 2020) attempt to address these concerns by aligning software-level models with known hardware-level fault characteristics.

Despite these limitations, software-based fault injection remains a critical part of the ML robustness research toolkit. When used appropriately, particularly when used in conjunction with hardware-based validation, these tools provide a scalable and efficient way to explore large design spaces, identify vulnerable components, and develop mitigation strategies. As fault modeling techniques continue to evolve, the integration of hardware-aware insights into software-based tools will be key to improving their realism and impact.

18.6.3.3 Tool Types

Over the past several years, software-based fault injection tools have been developed for a wide range of ML frameworks and use cases. These tools vary in their level of abstraction, target platforms, and the types of faults they can simulate. Many are built to integrate with popular machine learning libraries such as PyTorch and TensorFlow, making them accessible to researchers and practitioners already working within those ecosystems.

One of the earliest and most influential tools is Ares (Reagen et al. 2018), initially designed for the Keras framework. Developed at a time when deep neural networks (DNNs) were growing in popularity, Ares was one of the first tools to systematically explore the effects of hardware faults on DNNs. It provided support for injecting single-bit flips and evaluating bit-error rates (BER) across weights and activation values. Importantly, Ares was validated against a physical DNN accelerator implemented in silicon, demonstrating its relevance for hardware-level fault modeling. As the field matured, Ares was extended to support PyTorch, allowing researchers to analyze fault behavior in more modern ML settings.

Building on this foundation, PyTorchFI (Mahmoud et al. 2020) was introduced as a dedicated fault injection library for PyTorch. Developed in collaboration with Nvidia Research, PyTorchFI allows fault injection into key components of ML models, including weights, activations, and gradients. Its native support for GPU acceleration makes it especially well-suited for evaluating large models efficiently. As shown in Figure 18.41, even simple bit-level faults can cause severe visual and classification errors, including the appearance of 'phantom' objects in images, which could have downstream safety implications in domains like autonomous driving.

The modular and accessible design of PyTorchFI has led to its adoption in several follow-on projects. For example, PyTorchALFI (developed by Intel xColabs) extends PyTorchFI's capabilities to evaluate system-level safety in automotive applications. Similarly, Dr. DNA (D. Ma et al. 2024) from Meta introduces a

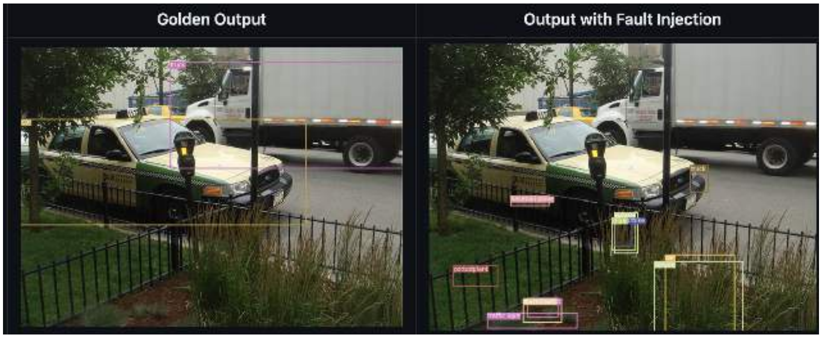


Figure 18.41: Hardware bitflips in ML workloads can cause phantom objects and misclassifications, which can erroneously be used downstream by larger systems, such as in autonomous driving. Shown above is a correct and faulty version of the same image using the PyTorchFI injection framework.

more streamlined, Pythonic API to simplify fault injection workflows. Another notable extension is GoldenEye (Mahmoud et al. 2022), which incorporates alternative numeric datatypes, including AdaptivFloat (Tambe et al. 2020) and BlockFloat, with bfloat16 as a specific example, to study the fault tolerance of non-traditional number formats under hardware-induced bit errors.

For researchers working within the TensorFlow ecosystem, TensorFI (Z. Chen et al. 2020) provides a parallel solution. Like PyTorchFI, TensorFI enables fault injection into the TensorFlow computational graph and supports a variety of fault models. One of TensorFI's strengths is its broad applicability—it can be used to evaluate many types of ML models beyond DNNs. Additional extensions such as BinFi (Z. Chen et al. 2019) aim to accelerate the fault injection process by focusing on the most critical bits in a model. This prioritization can help reduce simulation time while still capturing the most meaningful error patterns.

At a lower level of the software stack, NVBitFI (T. Tsai et al. 2021) offers a platform-independent tool for injecting faults directly into GPU assembly code. Developed by Nvidia, NVBitFI is capable of performing fault injection on any GPU-accelerated application, not just ML workloads. This makes it an especially powerful tool for studying resilience at the instruction level, where errors can propagate in subtle and complex ways. NVBitFI represents an important complement to higher-level tools like PyTorchFI and TensorFI, offering fine-grained control over GPU-level behavior and supporting a broader class of applications beyond machine learning.

Together, these tools offer a wide spectrum of fault injection capabilities. While some are tightly integrated with high-level ML frameworks for ease of use, others enable lower-level fault modeling with higher fidelity. By choosing the appropriate tool based on the level of abstraction, performance needs, and target application, researchers can tailor their studies to gain more actionable insights into the robustness of ML systems. The next section focuses on how these tools are being applied in domain-specific contexts, particularly in safety-critical systems such as autonomous vehicles and robotics.

18.6.3.4 Domain-Specific Examples

To address the unique challenges posed by specific application domains, researchers have developed specialized fault injection tools tailored to different machine learning (ML) systems. In high-stakes environments such as autonomous vehicles and robotics, domain-specific tools play a crucial role in evaluating system safety and reliability under hardware fault conditions. This section highlights three such tools: DriveFI and PyTorchALFI, which focus on autonomous vehicles, and MAVFI, which targets uncrewed aerial vehicles (UAVs). Each tool enables the injection of faults into mission-critical components, including perception, control, and sensor systems, providing researchers with insights into how hardware errors may propagate through real-world ML pipelines.

DriveFI (S. Jha et al. 2019) is a fault injection tool developed for autonomous vehicle systems. It facilitates the injection of hardware faults into the perception and control pipelines, enabling researchers to study how such faults affect system behavior and safety. Notably, DriveFI integrates with industry-standard platforms like Nvidia DriveAV and Baidu Apollo, offering a realistic environment for testing. Through this integration, DriveFI enables practitioners to evaluate the end-to-end resilience of autonomous vehicle architectures in the presence of fault conditions.

PyTorchALFI (Gräfe et al. 2023) extends the capabilities of PyTorchFI for use in the autonomous vehicle domain. Developed by Intel xColabs, PyTorchALFI enhances the underlying fault injection framework with domain-specific features. These include the ability to inject faults into multimodal sensor data[27], such as inputs from cameras and LiDAR systems. This allows for a deeper examination of how perception systems in autonomous vehicles respond to underlying hardware faults, further refining our understanding of system vulnerabilities and potential failure modes.

MAVFI (Hsiao et al. 2023) is a domain-specific fault injection framework tailored for robotics applications, particularly uncrewed aerial vehicles. Built atop the Robot Operating System (ROS), MAVFI provides a modular and extensible platform for injecting faults into various UAV subsystems, including sensors, actuators, and flight control algorithms. By assessing how injected faults impact flight stability and mission success, MAVFI offers a practical means for developing and validating fault-tolerant UAV architectures.

Together, these tools demonstrate the growing sophistication of fault injection research across application domains. By enabling fine-grained control over where and how faults are introduced, domain-specific tools provide actionable insights that general-purpose frameworks may overlook. Their development has greatly expanded the ML community's capacity to design and evaluate resilient systems—particularly in contexts where reliability, safety, and real-time performance are critical.

18.6.4 Bridging Hardware-Software Gap

While software-based fault injection tools offer many advantages in speed, flexibility, and accessibility, they do not always capture the full range of effects that hardware faults can impose on a system. This is largely due to the abstraction gap: software-based tools operate at a higher level and may overlook low-level hardware interactions or nuanced error propagation mechanisms that influence the behavior of ML systems in critical ways.

[27] Multimodal Sensor Data: Information collected simultaneously from multiple types of sensors (e.g., cameras, LiDAR, radar) to provide complementary perspectives of the environment. Critical for robust perception in autonomous systems.

As discussed in the work by (Bolchini et al. 2023), hardware faults can exhibit complex spatial distribution patterns that are difficult to replicate using purely software-based fault models. They identify four characteristic fault propagation patterns: single point, where the fault corrupts a single value in a feature map; same row, where a partial or entire row in a feature map is corrupted; bullet wake, where the same location across multiple feature maps is affected; and shatter glass, a more complex combination of both same row and bullet wake behaviors. These diverse patterns, visualized in Figure 18.42, highlight the limits of simplistic injection strategies and emphasize the need for hardware-aware modeling when evaluating ML system robustness.

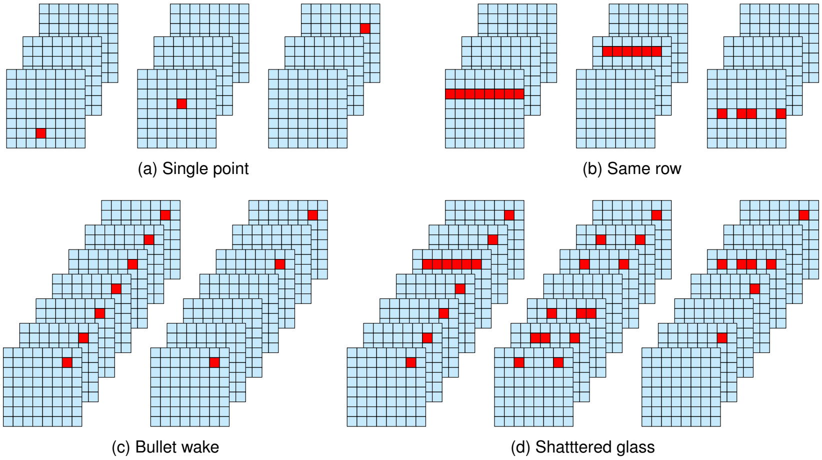


Figure 18.42: Hardware errors may manifest themselves in different ways at the software level, as classified by Bolchini et al. (2023). Spatial distribution patterns (erroneous values are colored in red). (a) *Single point:* The fault causes the corruption of a single value of a single feature map. (b) *Same row:* The fault causes the total or partial corruption of a row in a single feature map. (c) *Bullet wake:* The fault corrupts the same location in all or multiple feature maps. (d) *Shatter glass:* The fault causes the combination of the effects of same row and bullet wake patterns.

To address this abstraction gap, researchers have developed tools that explicitly aim to map low-level hardware error behavior to software-visible effects. One such tool is Fidelity, which bridges this gap by studying how hardware-level faults propagate and become observable at higher software layers. The next section discusses Fidelity in more detail.

18.6.4.1 Fidelity

Fidelity (Yi He, Balaprakash, and Li 2020) is a tool designed to model hardware faults more accurately within software-based fault injection experiments. Its core goal is to bridge the gap between low-level hardware fault behavior and the higher-level effects observed in machine learning systems by simulating how faults propagate through the compute stack.

The central insight behind Fidelity is that not all faults need to be modeled individually at the hardware level to yield meaningful results. Instead, Fidelity focuses on how faults manifest at the software-visible state and identifies equivalence relationships that allow representative modeling of entire fault classes. To accomplish this, it relies on several key principles:

First, fault propagation is studied to understand how a fault originating in hardware can move through various layers, including architectural registers, memory hierarchies, and numerical operations, eventually altering values in software. Fidelity captures these pathways to ensure that injected faults in software reflect the way faults would actually manifest in a real system.

Second, the tool identifies fault equivalence, which refers to grouping hardware faults that lead to similar observable outcomes in software. By focusing on representative examples rather than modeling every possible hardware bit flip individually, Fidelity allows more efficient simulations without sacrificing accuracy.

Finally, Fidelity uses a layered modeling approach, capturing the system's behavior at various abstraction levels—from hardware fault origin to its effect in the ML model's weights, activations, or predictions. This layering ensures that the impact of hardware faults is realistically simulated in the context of the ML system.

By combining these techniques, Fidelity allows researchers to run fault injection experiments that closely mirror the behavior of real hardware systems, but with the efficiency and flexibility of software-based tools. This makes Fidelity especially valuable in safety-critical settings, where the cost of failure is high and an accurate understanding of hardware-induced faults is essential.

18.6.4.2 Capturing Hardware Behavior

Capturing the true behavior of hardware faults in software-based fault injection tools is critical for advancing the reliability and robustness of ML systems. This fidelity becomes especially important when hardware faults have subtle but significant effects that may not be evident when modeled at a high level of abstraction.

There are several reasons why accurately reflecting hardware behavior is essential. First, accuracy is paramount. Software-based tools that mirror the actual propagation and manifestation of hardware faults provide more dependable insights into how faults influence model behavior. These insights are crucial for designing and validating fault-tolerant architectures and ensuring that mitigation strategies are grounded in realistic system behavior.

Second, reproducibility is improved when hardware effects are faithfully captured. This allows fault injection results to be reliably reproduced across different systems and environments, which is a cornerstone of rigorous scientific research. Researchers can better compare results, validate findings, and ensure consistency across studies.

Third, efficiency is enhanced when fault models focus on the most representative and impactful fault scenarios. Rather than exhaustively simulating every possible bit flip, tools can target a subset of faults that are known, by means of accurate modeling, to affect the system in meaningful ways. This selective approach saves computational resources while still providing comprehensive insights.

Finally, understanding how hardware faults appear at the software level is essential for designing effective mitigation strategies. When researchers know how specific hardware-level issues affect different components of an ML system, they can develop more targeted hardening techniques—such as retraining specific layers, applying redundancy selectively, or improving architectural resilience in bottleneck components.

Tools like Fidelity are central to this effort. By establishing mappings between low-level hardware behavior and higher-level software effects, Fidelity and similar tools empower researchers to conduct fault injection experiments that are not only faster and more scalable, but also grounded in real-world system behavior.

As ML systems continue to increase in scale and are deployed in increasingly safety-critical environments, this kind of hardware-aware modeling will become even more important. Ongoing research in this space aims to further refine the translation between hardware and software fault models and to develop tools that offer both efficiency and realism in evaluating ML system resilience. These advances will provide the community with more powerful, reliable methods for understanding and defending against the effects of hardware faults.

18.7 Conclusion

The pursuit of robust AI is a multifaceted endeavor that is critical for the reliable deployment of machine learning systems in real-world environments. As ML move from controlled research settings to practical applications, robustness becomes not just a desirable feature but a foundational requirement. Deploying AI in practice means engaging directly with the challenges that can compromise system performance, safety, and reliability.

We examined the broad spectrum of issues that threaten AI robustness, beginning with hardware-level faults. Transient faults may introduce temporary computational errors, while permanent faults, including the well-known Intel FDIV bug, can lead to persistent inaccuracies that affect system behavior over time.

Beyond hardware, machine learning models themselves are susceptible to a variety of threats. Adversarial examples, such as the misclassification of modified stop signs, reveal how subtle input manipulations can cause erroneous outputs. Likewise, data poisoning techniques, exemplified by the Nightshade project, illustrate how malicious training data can degrade model performance or implant hidden backdoors, posing serious security risks in practical deployments.

The chapter also addressed the impact of distribution shifts, which often result from temporal evolution or domain mismatches between training and deployment environments. Such shifts challenge a model's ability to generalize and perform reliably under changing conditions. Compounding these issues are faults in the software infrastructure, including frameworks, libraries, and runtime components, which can propagate unpredictably and undermine system integrity.

To navigate these risks, the use of robust tools and evaluation frameworks is essential. Tools such as PyTorchFI and Fidelity enable researchers and practitioners to simulate fault scenarios, assess vulnerabilities, and systematically improve system resilience. These resources are critical for translating theoretical robustness principles into operational safeguards.

Ultimately, building robust AI requires a comprehensive and proactive approach. Fault tolerance, security mechanisms, and continuous monitoring must be embedded throughout the AI development lifecycle—from data collection and model training to deployment and maintenance. As this chapter has demonstrated, applying AI in real-world contexts means addressing these robustness challenges head-on to ensure that systems operate safely, reliably, and effectively in complex and evolving environments.

18.8 Resources

ℹ Slides

- *Coming soon.*

! Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 19

AI for Good

Figure 19.1: *DALL·E 3 Prompt: Illustration of planet Earth wrapped in shimmering neural networks, with diverse humans and AI robots working together on various projects like planting trees, cleaning the oceans, and developing sustainable energy solutions. The positive and hopeful atmosphere represents a united effort to create a better future.*

Purpose

How can we harness machine learning systems to address critical societal challenges, and what principles guide the development of solutions that create lasting positive impact?

The application of AI systems to societal challenges represents the culmination of technical capability and social responsibility. Impact-driven development reveals essential patterns for translating technological potential into meaningful change, highlighting critical relationships between system design and societal outcomes. The implementation of solutions for social good showcases pathways for addressing complex challenges while maintaining technical rigor and operational effectiveness. Understanding these impact dynamics provides insights into creating transformative systems, establishing principles for designing AI solutions that advance human welfare, and promote positive societal transformation.

Learning Objectives

- Explore how AI systems can address critical real-world societal challenges.
- Recognize key design patterns for ML systems in social impact.
- Select suitable design patterns based on resource availability and adaptability needs.
- Explore how Cloud ML, Edge ML, Mobile ML, and Tiny ML integrate into these patterns.
- Evaluate the strengths and limitations of design patterns for specific deployment scenarios.

19.1 Overview

Previous chapters examined the fundamental components of machine learning systems - from neural architectures and training methodologies to acceleration techniques and deployment strategies. These chapters established how to build, optimize, and operate ML systems at scale. The examples and techniques focused primarily on scenarios where computational resources, reliable infrastructure, and technical expertise were readily available.

Machine learning systems, however, extend beyond commercial and industrial applications. While recommendation engines, computer vision systems, and natural language processors drive business value, ML systems also hold immense potential for addressing pressing societal challenges. This potential remains largely unrealized due to the distinct challenges of deploying ML systems in resource-constrained environments[0].

Engineering ML systems for social impact differs fundamentally from commercial deployments. These systems must operate in environments with limited computing resources, intermittent connectivity, and minimal technical support infrastructure. Such constraints reshape every aspect of ML system design—from model architecture and training approaches to deployment patterns and maintenance strategies. Success requires rethinking traditional ML system design patterns to create solutions that are robust, maintainable, and effective despite these limitations.

Building ML systems for AI for social good is an engineering challenge.

Definition of AI for Good

AI for Good refers to the *design, development, and deployment of machine learning systems aimed at addressing critical societal and environmental challenges*. These systems seek to *enhance human welfare, promote sustainability, and contribute to global development goals* by leveraging machine learning and related AI technologies to *create positive, equitable, and lasting impact*.

[0] Resource-constrained environments: Areas with limited computing capabilities, connectivity, and support infrastructure.

This chapter highlights some AI applications for social good and examines the unique requirements, constraints, and opportunities in engineering ML systems for social impact. We analyze how core ML system components adapt to resource-constrained environments, explore architectural patterns that enable robust deployment across the computing spectrum, and study real-world implementations in healthcare, agriculture, education, and environmental monitoring. Through these examples and the discussions involved, we develop frameworks for designing ML systems that deliver sustainable social impact.

19.2 Global Challenges

History provides sobering examples of where timely interventions and coordinated responses could have dramatically altered outcomes. The 2014-2016 Ebola outbreak in West Africa, for instance, highlighted the catastrophic consequences of delayed detection and response systems (WHO). Similarly, the 2011 famine in Somalia, despite being forecasted months in advance, caused immense suffering due to inadequate mechanisms to mobilize and allocate resources effectively (ReliefWeb). In the aftermath of the 2010 Haiti earthquake, the lack of rapid and reliable damage assessment significantly hampered efforts to direct aid where it was most needed (USGS).

Today, similar challenges persist across diverse domains, particularly in resource-constrained environments. In healthcare, remote and underserved communities often experience preventable health crises due to the absence of timely access to medical expertise. A lack of diagnostic tools and specialists means that treatable conditions can escalate into life-threatening situations, creating unnecessary suffering and loss of life. Agriculture, a sector critical to global food security, faces parallel struggles. Smallholder farmers, responsible for producing much of the world's food, make crucial decisions with limited information. Increasingly erratic weather patterns, pest outbreaks, and soil degradation compound their difficulties, often resulting in reduced yields and heightened food insecurity, particularly in vulnerable regions. These challenges demonstrate how systemic barriers and resource constraints perpetuate inequities and undermine resilience.

Similar systemic barriers are evident in education, where inequity further amplifies challenges in underserved areas. Many schools lack sufficient teachers, adequate resources, and personalized support for students. This not only widens the gap between advantaged and disadvantaged learners but also creates long-term consequences for social and economic development. Without access to quality education, entire communities are left at a disadvantage, perpetuating cycles of poverty and inequality. These inequities are deeply interconnected with broader challenges, as gaps in education often exacerbate issues in other critical sectors such as healthcare and agriculture.

The strain on ecosystems introduces another dimension to these challenges. Environmental degradation, including deforestation, pollution, and biodiversity loss, threatens livelihoods and destabilizes the ecological balance necessary for sustaining human life. Vast stretches of forests, oceans, and wildlife habitats remain unmonitored and unprotected, particularly in regions with limited resources. This leaves ecosystems vulnerable to illegal activities such as poaching, logging, and pollution, further intensifying the pressures on communities already grappling with economic and social disparities. These interwoven challenges underscore the need for holistic solutions that address both human and environmental vulnerabilities.

Although these issues vary in scope and scale, they share several critical characteristics. They disproportionately affect vulnerable populations, exacerbating existing inequalities. Resource constraints in affected regions pose significant barriers to implementing solutions. Moreover, addressing these challenges requires navigating trade-offs between competing priorities and limited resources, often under conditions of great uncertainty.

Technology holds the potential to play a transformative role in addressing these issues. By providing innovative tools to enhance decision-making, increase efficiency, and deliver solutions at scale, it offers hope for overcoming the barriers that have historically hindered progress. Among these technologies, machine learning systems stand out for their capacity to process vast amounts of information, uncover patterns, and generate insights that can inform action in even the most resource-constrained environments. However, realizing this potential requires deliberate and systematic approaches to ensure these tools are designed and implemented to serve the needs of all communities effectively and equitably.

19.3 Key AI Applications

AI technologies, such as Cloud ML, Mobile ML, Edge ML, and Tiny ML, are unlocking transformative solutions to some of the world's most pressing challenges. By adapting to diverse constraints and leveraging unique strengths, these technologies are driving innovation in agriculture, healthcare, disaster response, and environmental conservation. This section explores how these paradigms bring social good to life through real-world applications.

19.3.1 Agriculture

Watch on YouTube
Plant Village Nuru

Scan with your phone
to watch the video

! Important 11: Plant Village Nuru

🖥 Watch on YouTube

In Sub-Saharan Africa, cassava farmers have long battled diseases that devastate crops and livelihoods. Now, with the help of mobile ML-powered smartphone apps, as shown in Figure 19.2, they can snap a photo of a leaf and receive instant feedback on potential diseases. This early detection system has reduced cassava losses from 40% to just 5%, offering hope to farmers in disconnected regions where access to agricultural advisors is limited (Ramcharan et al. 2017).

Across Southeast Asia, rice farmers are confronting increasingly unpredictable weather patterns. In Indonesia, Tiny ML sensors are transforming their ability to adapt by monitoring microclimates across paddies. These low-power devices process data locally to optimize water usage, enabling precision irrigation even in areas with minimal infrastructure (Tirtalistyani, Murtiningrum, and Kanwar 2022).

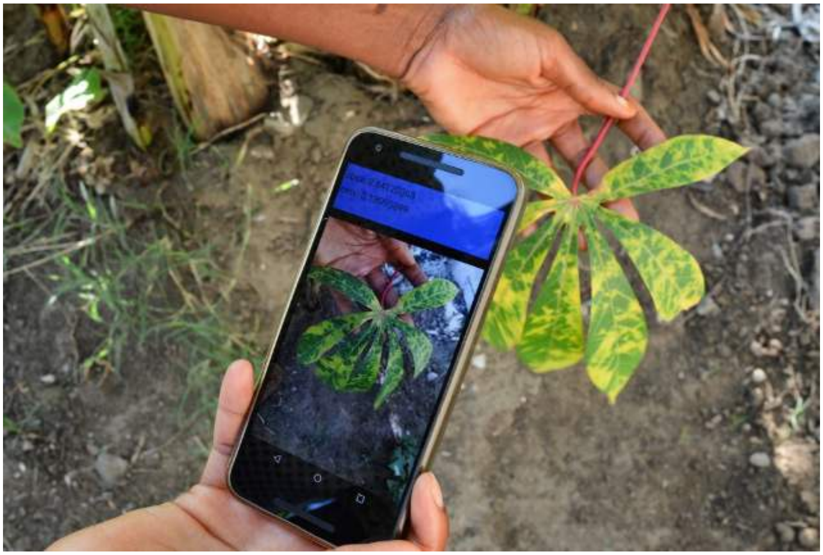


Figure 19.2: AI helps farmers to detect plant diseases.

On a global scale, Microsoft's FarmBeats is pioneering the integration of IoT sensors, drones, and Cloud ML to create actionable insights for farmers. By leveraging weather forecasts, soil conditions, and crop health data, the platform allows farmers to optimize inputs like water and fertilizer, reducing waste and improving yields. Together, these innovations illustrate how AI technologies are bringing precision agriculture to life, addressing food security, sustainability, and climate resilience.

19.3.2 Healthcare

For millions in underserved communities, access to healthcare often means long waits and travel to distant clinics. Tiny ML is changing that by enabling diagnostics to occur at the patient's side. For example, a low-cost wearable developed by Respira x Colabs uses embedded machine learning to analyze cough patterns and detect pneumonia. Designed for remote areas, the device operates independently of internet connectivity and is powered by a simple microcontroller, making life-saving diagnostics accessible to those who need it most.

Tiny ML's potential extends to tackling global health issues like vector-borne diseases[1] that are spread by mosquitoes. Researchers have developed low-cost devices that use machine learning to identify mosquito species by their wingbeat frequencies (Altayeb, Zennaro, and Rovai 2022). This technology enables real-time monitoring of malaria-carrying mosquitoes. It offers a scalable solution for malaria control in high-risk regions.

In parallel, Cloud ML is advancing healthcare research and diagnostics on a broader scale. Platforms like Google Genomics analyze vast datasets to identify disease markers, accelerating breakthroughs in personalized medicine. These examples show how AI technologies, ranging from the portability of Tiny

[1] Vector-borne diseases: Illnesses caused by pathogens transmitted by vectors like mosquitoes, ticks, or fleas.

ML to the computational power of Cloud ML, are converging to democratize healthcare access and improve outcomes worldwide.

19.3.3 Disaster Response

In disaster zones, where every second counts, AI technologies are providing tools to accelerate response efforts and enhance safety. Tiny, autonomous drones equipped with Tiny ML algorithms are making their way into collapsed buildings, navigating obstacles to detect signs of life. By analyzing thermal imaging and acoustic signals locally, these drones can identify survivors and hazards without relying on cloud connectivity (Duisterhof et al. 2021). Video 12 and Video 13 show how Tiny ML algorithms can be used to enable drones to autonomously seek light and gas sources.

Watch on YouTube
Light Seeking

Gas Seeking

Scan with your phone
to watch the video

! Important 12: Light Seeking

🖥 Watch on YouTube

! Important 13: Gas Seeking

🖥 Watch on YouTube

At a broader level, platforms like Google's AI for Disaster Response are leveraging Cloud ML to process satellite imagery and predict flood zones. These systems provide real-time insights to help governments allocate resources more effectively and save lives during emergencies.

Mobile ML applications are also playing a critical role by delivering real-time disaster alerts directly to smartphones. Tsunami warnings and wildfire updates tailored to users' locations enable faster evacuations and better preparedness. Whether scaling globally with Cloud ML or enabling localized insights with Edge and Mobile ML, these technologies are redefining disaster response capabilities.

19.3.4 Environmental Conservation

Conservationists face immense challenges in monitoring and protecting biodiversity across vast and often remote landscapes. AI technologies are offering scalable solutions to these problems, combining local autonomy with global coordination.

Watch on YouTube
Elephant Edge

Scan with your phone
to watch the video

! Important 14: Elephant Edge

🖥 Watch on YouTube

EdgeML-powered collars are being used to unobtrusively track animal behavior, such as elephant movements and vocalizations (Video 14). By processing data on the collar itself, these devices minimize power consumption and reduce the need for frequent battery changes (Verma 2022). Meanwhile, Tiny ML systems are enabling anti-poaching efforts by detecting threats like gunshots or human activity and relaying alerts to rangers in real time (Bamoumen et al. 2022).

At a global scale, Cloud ML is being used to monitor illegal fishing activities. Platforms like Global Fishing Watch analyze satellite data to detect anomalies, helping governments enforce regulations and protect marine ecosystems. These examples highlight how AI technologies are enabling real-time monitoring and decision-making, advancing conservation efforts in profound ways.

19.3.5 AI's Holistic Impact

The examples highlighted above demonstrate the transformative potential of AI technologies in addressing critical societal challenges. However, these successes also underscore the complexity of tackling such problems holistically. Each example addresses specific needs, such as optimizing agricultural resources, expanding healthcare access, or protecting ecosystems, but solving these issues sustainably requires more than isolated innovations.

To maximize impact and ensure equitable progress, collective efforts are essential. Large-scale challenges demand collaboration across sectors, geographies, and stakeholders. By fostering coordination between local initiatives, research institutions, and global organizations, we can align AI's transformative potential with the infrastructure and policies needed to scale solutions effectively. Without such alignment, even the most promising innovations risk operating in silos, limiting their reach and long-term sustainability.

To address this, we require frameworks that help harmonize efforts and prioritize initiatives that deliver broad, lasting impact. These frameworks serve as roadmaps to bridge the gap between technological potential and meaningful global progress.

19.4 Global Development Perspective

The sheer scale and complexity of these problems demand a systematic approach to ensure that efforts are targeted, coordinated, and sustainable. This is where global frameworks such as the United Nations Sustainable Development Goals (SDGs) and guidance from institutions like the World Health Organization (WHO)[2] play a pivotal role. These frameworks provide a structured lens for thinking about and addressing the world's most pressing challenges. They offer a roadmap to align efforts, set priorities, and foster international collaboration to create impactful and lasting change (*The Sustainable Development Goals Report 2018* 2018).

The SDGs shown in Figure 19.3 are a global agenda adopted in 2015. These 17 interconnected goals form a blueprint for addressing the world's most pressing challenges by 2030. They range from eliminating poverty and hunger to ensuring quality education, from promoting gender equality to taking climate action.

Machine learning systems can contribute to multiple SDGs simultaneously through their transformative capabilities:

[2] World Health Organization (WHO): A specialized agency of the United Nations responsible for international public health.

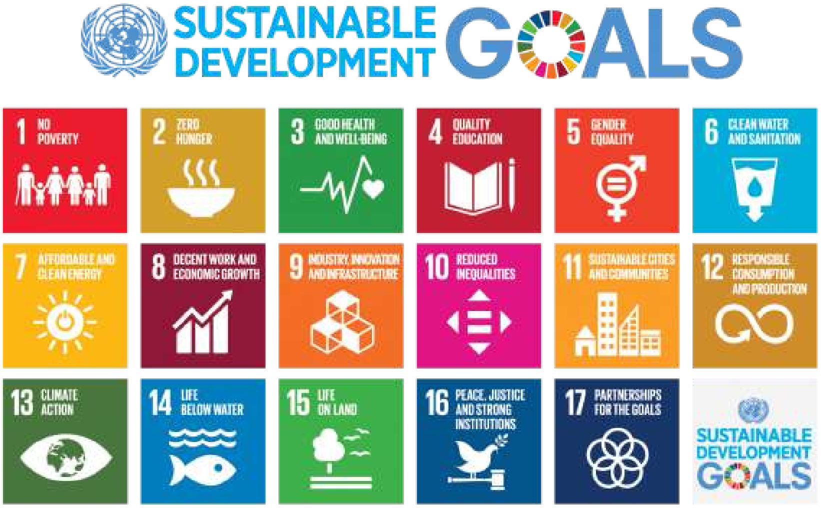

SUSTAINABLE DEVELOPMENT GOALS
1 NO POVERTY
2 ZERO HUNGER
3 GOOD HEALTH AND WELL-BEING
4 QUALITY EDUCATION
5 GENDER EQUALITY
6 CLEAN WATER AND SANITATION
7 AFFORDABLE AND CLEAN ENERGY
8 DECENT WORK AND ECONOMIC GROWTH
9 INDUSTRY, INNOVATION AND INFRASTRUCTURE
10 REDUCED INEQUALITIES
11 SUSTAINABLE CITIES AND COMMUNITIES
12 RESPONSIBLE CONSUMPTION AND PRODUCTION
13 CLIMATE ACTION
14 LIFE BELOW WATER
15 LIFE ON LAND
16 PEACE, JUSTICE AND STRONG INSTITUTIONS
17 PARTNERSHIPS FOR THE GOALS
SUSTAINABLE DEVELOPMENT GOALS

Figure 19.3: United Nations Sustainable Development Goals (SDG). Source: United Nations.

- **Goal 1 (No Poverty) & Goal 10 (Reduced Inequalities)**: ML systems that improve financial inclusion through mobile banking and risk assessment for microloans.
- **Goals 2, 12, & 15 (Zero Hunger, Responsible Consumption, Life on Land)**: Systems that optimize resource distribution, reduce waste in food supply chains, and monitor biodiversity.
- **Goals 3 & 5 (Good Health and Gender Equality)**: ML applications that improve maternal health outcomes and access to healthcare in underserved communities.
- **Goals 13 & 11 (Climate Action & Sustainable Cities)**: Predictive systems for climate resilience and urban planning that help communities adapt to environmental changes.

However, deploying these systems presents unique challenges. Many regions that could benefit most from machine learning applications lack reliable electricity (Goal 7: Affordable and Clean Energy) or internet infrastructure (Goal 9: Industry, Innovation and Infrastructure). This reality forces us to rethink how we design machine learning systems for social impact.

Success in advancing the SDGs through machine learning requires a holistic approach that goes beyond technical solutions. Systems must operate within local resource constraints while respecting cultural contexts and existing infrastructure limitations. This reality pushes us to fundamentally rethink system design, considering not just technological capabilities but also their sustainable integration into communities that need them most.

The following sections explore how to navigate these technical, infrastructural, and societal factors to create ML systems that genuinely advance sustainable development goals without creating new dependencies or deepening existing inequalities.

19.5 Engineering Challenges

Deploying machine learning systems in social impact contexts requires us to navigate a series of interconnected challenges spanning computational, networking, power, and data dimensions. These challenges are particularly pronounced when transitioning from development to production environments or scaling deployments in resource-constrained settings.

To provide an overview, Table 19.1 summarizes the key differences in resources and requirements across development, rural, and urban contexts, while also highlighting the unique constraints encountered during scaling. This comparison provides a basis for understanding the paradoxes, dilemmas, and constraints that will be explored in subsequent sections.

Table 19.1: Comparison of resource constraints and challenges across rural deployments, urban deployments, and scaling in machine learning systems for social impact contexts.

Aspect	Rural Deployment	Urban Deployment	Scaling Challenges
Computational Resources	Microcontroller (ESP32: 240 MHz, 520 KB RAM)	Server-grade systems (100-200 W, 32-64 GB RAM)	Aggressive model quantization (e.g., 50 MB to 500 KB)
Power Infrastructure	Solar and battery systems (10-20 W, 2000-3000 mAh battery)	Stable grid power	Optimized power usage (for deployment devices)
Network Bandwidth	LoRa, NB-IoT (0.3-50 kbps, 60-250 kbps)	High-bandwidth options	Protocol adjustments (LoRa, NB-IoT, Sigfox: 100-600 bps)
Data Availability	Sparse, heterogeneous data sources (500 KB/day from rural clinics)	Large volumes of standardized data (Gigabytes from urban hospitals)	Specialized pipelines (For privacy-sensitive data)
Model Footprint	Highly quantized models (≤ 1 MB)	Cloud/edge systems (Supporting larger models)	Model architecture redesign (For size, power, and bandwidth limits)

19.5.1 Resource Paradox

Deploying machine learning systems in social impact contexts reveals a fundamental resource paradox that shapes every aspect of system design. While areas with the greatest needs could benefit most from machine learning capabilities, they often lack the basic infrastructure required for traditional deployments.

This paradox becomes evident in the computational and power requirements of machine learning systems, as shown in Table 19.1. A typical cloud deployment might utilize servers consuming 100-200 W of power with multiple CPU cores and 32-64 GB of RAM. However, rural deployments must often operate on single-board computers drawing 5 W or microcontrollers consuming mere milliwatts, with RAM measured in kilobytes rather than gigabytes.

Network infrastructure limitations further constrain system design. Urban environments offer high-bandwidth options like fiber (100+ Mbps) and 5G networks (1-10 Gbps). Rural deployments must instead rely on low-power wide-area network technologies such as LoRa or NB-IoT[3], which achieve kilometer-range coverage with minimal power consumption.

Power infrastructure presents additional challenges. While urban systems can rely on stable grid power, rural deployments often depend on solar charging

[3] NB-IoT (Narrowband Internet of Things): NB-IoT is a low-power, wide-area wireless communication technology optimized for connecting IoT devices with minimal energy usage, often in resource-constrained environments.

and battery systems. A typical solar-powered system might generate 10-20 W during peak sunlight hours, requiring careful power budgeting across all system components. Battery capacity limitations, often 2000-3000 mAh, mean systems must optimize every aspect of operation, from sensor sampling rates to model inference frequency.

19.5.2 Data Dilemma

Beyond just computational horserpower, machine learning systems in social impact contexts face fundamental data challenges that differ significantly from commercial deployments. Where commercial systems often work with standardized datasets containing millions of examples, social impact projects must build robust systems with limited, heterogeneous data sources.

Healthcare deployments illustrate these data constraints clearly. A typical rural clinic might generate 50-100 patient records per day, combining digital entries with handwritten notes. These records often mix structured data like vital signs with unstructured observations, requiring specialized preprocessing pipelines. The total data volume might reach only 500 KB per day. This is a stark contrast to urban hospitals generating gigabytes of standardized electronic health records. Even an X-ray or MRI scan is measured in megabytes or more, underscoring the vast disparity in data scales between rural and urban healthcare facilities.

Network limitations further constrain data collection and processing. Agricultural sensor networks, operating on limited power budgets, might transmit only 100-200 bytes per reading. With LoRa[4] bandwidth constraints of 50 kbps, these systems often limit transmission frequency to once per hour. A network of 1000 sensors thus generates only 4-5 MB of data per day, requiring models to learn from sparse temporal data. For perspective, streaming a single minute of video on Netflix can consume several megabytes, highlighting the disparity in data volumes between industrial IoT networks and everyday internet usage.

Privacy considerations add another layer of complexity. Protecting sensitive information while operating within hardware constraints requires careful system design. Implementing privacy-preserving techniques on devices with 512 KB RAM means partitioning models and data carefully. Local processing must balance privacy requirements against hardware limitations, often restricting model sizes to under 1 MB. Supporting multiple regional variants of these models can quickly consume the limited storage available on low-cost devices, typically 2-4 MB total.

19.5.3 Scale Challenge

Scaling machine learning systems from prototype to production deployment introduces fundamental resource constraints that necessitate architectural redesign. Development environments provide computational resources that mask many real-world limitations. A typical development platform, such as a Raspberry Pi 4, offers substantial computing power with its 1.5 GHz processor and 4 GB RAM. These resources enable rapid prototyping and testing of machine learning models without immediate concern for optimization.

[4] LoRa (Long Range): LoRA is a low-power wireless communication protocol designed for transmitting small data packets over long distances with minimal energy consumption.

Production deployments reveal stark resource limitations. When scaling to thousands of devices, cost and power constraints often mandate the use of microcontroller units like the ESP32, a widely used microcontroller unit from Espressif Systems, with its 240 MHz processor and mere 520 KB of RAM. This dramatic reduction in computational resources demands fundamental changes in system architecture. Models must be redesigned, optimization techniques such as quantization and pruning applied, and inference strategies reconsidered.

Network infrastructure constraints fundamentally influence system architecture at scale. Different deployment contexts necessitate different communication protocols, each with distinct operational parameters. This heterogeneity in network infrastructure requires systems to maintain consistent performance across varying bandwidth and latency conditions. As deployments scale across regions, system architectures must accommodate seamless transitions between network technologies while preserving functionality.

The transformation from development to scaled deployment presents consistent patterns across application domains. Environmental monitoring systems exemplify these scaling requirements. A typical forest monitoring system might begin with a 50 MB computer vision model running on a development platform. Scaling to widespread deployment necessitates reducing the model to approximately 500 KB through quantization and architectural optimization, enabling operation on distributed sensor nodes. This reduction in model footprint must preserve detection accuracy while operating within strict power constraints of 1-2 W. Similar architectural transformations occur in agricultural monitoring systems and educational platforms, where models must be optimized for deployment across thousands of resource-constrained devices while maintaining system efficacy.

19.5.4 Sustainability Challenge

Maintaining machine learning systems in resource-constrained environments presents distinct challenges that extend beyond initial deployment considerations. These challenges encompass system longevity, environmental impact, community capacity, and financial viability—factors that ultimately determine the long-term success of social impact initiatives.

System longevity requires careful consideration of hardware durability and maintainability. Environmental factors such as temperature variations (typically -20°C to 50°C in rural deployments), humidity (often 80-95% in tropical regions), and dust exposure significantly impact component lifetime. These conditions necessitate robust hardware selection and protective measures that balance durability against cost constraints. For instance, solar-powered agricultural monitoring systems must maintain consistent operation despite seasonal variations in solar irradiance[5], typically ranging from 3-7 kWh/m^2/day depending on geographical location and weather patterns.

Environmental sustainability introduces additional complexity in system design. The environmental footprint of deployed systems includes not only operational power consumption but also the impact of manufacturing, transportation, and end-of-life disposal, which we had discussed in previous chapters.

[5] Solar Irradiance: The power per unit area received from the Sun in the form of electromagnetic radiation, typically measured in watts per square meter (W/m^2). It varies with geographic location, time of day, and atmospheric conditions.

A typical deployment of 1000 sensor nodes requires consideration of approximately 500 kg of electronic components, including sensors, processing units, and power systems. Sustainable design principles must address both immediate operational requirements and long-term environmental impact through careful component selection and end-of-life planning.

Community capacity building represents another critical dimension of sustainability. Systems must be maintainable by local technicians with varying levels of expertise. This requirement influences architectural decisions, from component selection to system modularity. Documentation must be comprehensive yet accessible, typically requiring materials in multiple languages and formats. Training programs must bridge knowledge gaps while building local technical capacity, ensuring that communities can independently maintain and adapt systems as needs evolve.

Financial sustainability often determines system longevity. Operating costs, including maintenance, replacement parts, and network connectivity, must align with local economic conditions. A sustainable deployment might target operational costs below 5% of local monthly income per beneficiary. This constraint influences every aspect of system design, from hardware selection to maintenance schedules, requiring careful optimization of both capital and operational expenditures.

19.6 Design Patterns

The challenges of deploying machine learning systems in resource-constrained environments reflect fundamental constraints that have shaped system architecture for decades. Computing systems across domains have developed robust solutions to operate within limited computational resources, unreliable networks, and power restrictions. These solutions, formalized as "design patterns[6]," represent reusable architectural approaches to common deployment challenges.

Traditional system design patterns from distributed systems, embedded computing, and mobile applications provide valuable frameworks for machine learning deployments. The Hierarchical Processing Pattern, for instance, structures operations across system tiers to optimize resource usage. Progressive enhancement ensures graceful degradation under varying conditions, while the Distributed Knowledge Pattern sharing enables consistency across multiple data sources. These established patterns can be adapted to address the unique requirements of machine learning systems, particularly regarding model deployment, training procedures, and inference operations.

19.6.1 Hierarchical Processing

The Hierarchical Processing Pattern organizes systems into tiers that share responsibilities based on their available resources and capabilities. Like a business with local branches, regional offices, and headquarters, this pattern segments workloads across edge, regional, and cloud tiers. Each tier is optimized for specific tasks—edge devices handle data collection and local processing, regional nodes[7] manage aggregation and intermediate computations, and cloud

[6] Reusable architectural approaches to address common challenges in system design and deployment.

[7] Regional nodes: Intermediate computing layers that consolidate and preprocess data between edge and cloud tiers.

infrastructure supports advanced analytics and model training.

Figure 19.4 depicts the interaction flow across these tiers. Starting at the edge tier with data collection, information flows through regional aggregation and processing, culminating in cloud-based advanced analysis. Bidirectional feedback loops enable model updates to flow back through the hierarchy, ensuring continuous system improvement.

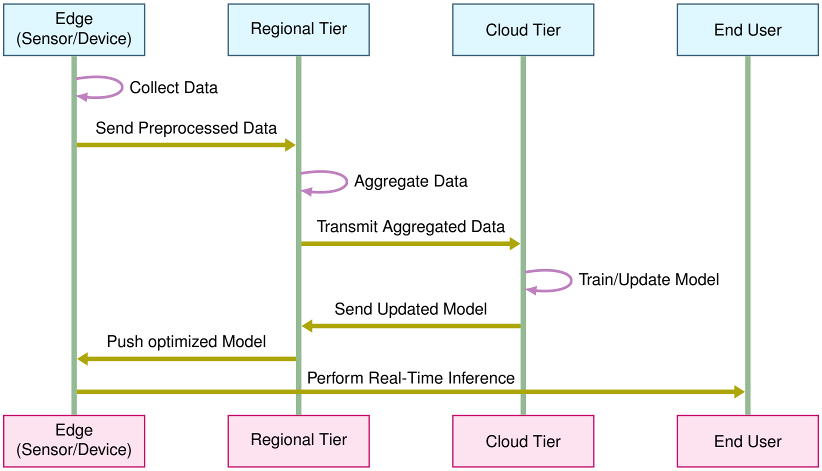


Figure 19.4: Sequence diagram illustrating the Hierarchical Processing Pattern.

This architecture excels in environments with varying infrastructure quality, such as applications spanning urban and rural regions. Edge devices maintain critical functionalities during network or power disruptions by performing essential computations locally while queuing operations that require higher-tier resources. When connectivity returns, the system scales operations across available infrastructure tiers.

In machine learning applications, this pattern requires careful consideration of resource allocation and data flow. Edge devices must balance model inference accuracy against computational constraints, while regional nodes facilitate data aggregation and model personalization. Cloud infrastructure provides the computational power needed for comprehensive analytics and model retraining. This distribution demands thoughtful optimization of model architectures, training procedures, and update mechanisms throughout the hierarchy.

19.6.1.1 Google's Flood Forecasting

❗ Important 15: AI for Flood Forecasting

🖵 Watch on YouTube

Watch on YouTube
AI for Flood Forecasting

Scan with your phone
to watch the video

Google's Flood Forecasting Initiative demonstrates how the Hierarchical Processing Pattern supports large-scale environmental monitoring. Edge devices along river networks monitor water levels, performing basic anomaly detection even without cloud connectivity. Regional centers aggregate this data and ensure localized decision-making, while the cloud tier integrates inputs from multiple regions for advanced flood prediction and system-wide updates. This tiered approach balances local autonomy with centralized intelligence, ensuring functionality across diverse infrastructure conditions[8].

At the edge tier, the system likely employs water-level sensors and local processing units distributed along river networks. These devices perform two critical functions: continuous monitoring of water levels at regular intervals (e.g., every 15 minutes) and preliminary time-series analysis to detect significant changes. Constrained by the tight power envelope (a few watts of power), edge devices utilize quantized models for anomaly detection, enabling low-power operation and minimizing the volume of data transmitted to higher tiers. This localized processing ensures that key monitoring tasks can continue independently of network connectivity.

The regional tier operates at district-level processing centers, each responsible for managing data from hundreds of sensors across its jurisdiction. At this tier, more sophisticated neural network models are employed to combine sensor data with additional contextual information, such as local terrain features and historical flood patterns. This tier reduces the data volume transmitted to the cloud by aggregating and extracting meaningful features while maintaining critical decision-making capabilities during network disruptions. By operating independently when required, the regional tier enhances system resilience and ensures localized monitoring and alerts remain functional.

At the cloud tier, the system integrates data from regional centers with external sources such as satellite imagery and weather data to implement the full machine learning pipeline. This includes training and running advanced flood prediction models, generating inundation maps, and distributing predictions to stakeholders. The cloud tier provides the computational resources needed for large-scale analysis and system-wide updates. However, the hierarchical structure ensures that essential monitoring and alerting functions can continue autonomously at the edge and regional tiers, even when cloud connectivity is unavailable.

This implementation reveals several key principles of successful Hierarchical Processing Pattern deployments. First, the careful segmentation of ML tasks across tiers enables graceful degradation. Each tier maintains critical functionality even when isolated. Secondly, the progressive enhancement of capabilities as higher tiers become available demonstrates how systems can adapt to varying resource availability. Finally, the bidirectional flow of information, where sensor data moves upward and model updates flow downward, creates a robust feedback loop that improves system performance over time. These principles extend beyond flood forecasting to inform hierarchical ML deployments across various social impact domains.

19.6.1.2 Structure

The Hierarchical Processing Pattern implements specific architectural components and relationships that enable its distributed operation. Understanding these structural elements is crucial for effective implementation across different deployment scenarios.

[8] Google's Flood Forecasting Initiative has been instrumental in mitigating flood risks in vulnerable regions, including parts of India and Bangladesh. By combining real-time sensor data with machine learning models, the initiative generates precise flood predictions and timely alerts, reducing disaster-related losses and enhancing community preparedness.

The edge tier's architecture centers on resource-aware components that optimize local processing capabilities. At the hardware level, data acquisition modules implement adaptive sampling rates, typically ranging from 1 Hz to 0.01 Hz, adjusting dynamically based on power availability. Local storage buffers, usually 1-4 MB, manage data during network interruptions through circular buffer implementations. The processing architecture incorporates lightweight inference engines specifically optimized for quantized models, working alongside state management systems that continuously track device health and resource utilization. Communication modules implement store-and-forward protocols designed for unreliable networks, ensuring data integrity during intermittent connectivity.

The regional tier implements aggregation and coordination structures that enable distributed decision-making. Data fusion engines are the core of this tier, combining multiple edge data streams while accounting for temporal and spatial relationships. Distributed databases, typically spanning 50-100 GB, support eventual consistency models to maintain data coherence across nodes. The tier's architecture includes load balancing systems that dynamically distribute processing tasks based on available computational resources and network conditions. Failover mechanisms ensure continuous operation during node failures, while model serving infrastructure supports multiple model versions to accommodate varying edge device capabilities. Inter-region synchronization protocols manage data consistency across geographic boundaries.

The cloud tier provides the architectural foundation for system-wide operations through sophisticated distributed systems. Training infrastructure supports parallel model updates across multiple compute clusters, while version control systems manage model lineage and deployment histories. High-throughput data pipelines process incoming data streams from all regional nodes, implementing automated quality control and validation mechanisms. The architecture includes robust security frameworks that manage authentication and authorization across all tiers while maintaining audit trails of system access and modifications. Global state management systems track the health and performance of the entire deployment, enabling proactive resource allocation and system optimization.

The Hierarchical Processing Pattern's structure enables sophisticated management of resources and responsibilities across tiers. This architectural approach ensures that systems can maintain critical operations under varying conditions while efficiently utilizing available resources at each level of the hierarchy.

19.6.1.3 Modern Adaptations

Advancements in computational efficiency, model design, and distributed systems have transformed the traditional Hierarchical Processing Pattern. While maintaining its core principles, the pattern has evolved to accommodate new technologies and methodologies that enable more complex workloads and dynamic resource allocation. These innovations have particularly impacted how the different tiers interact and share responsibilities, creating more flexible and capable deployments across diverse environments.

One of the most notable transformations has occurred at the edge tier. Historically constrained to basic operations such as data collection and simple preprocessing, edge devices now perform sophisticated processing tasks that were previously exclusive to the cloud. This shift has been driven by two critical developments: efficient model architectures and hardware acceleration. Techniques such as model compression, pruning, and quantization have dramatically reduced the size and computational requirements of neural networks, allowing even resource-constrained devices to perform inference tasks with reasonable accuracy. Advances in specialized hardware, such as edge AI accelerators and low-power GPUs, have further enhanced the computational capabilities of edge devices. As a result, tasks like image recognition or anomaly detection that once required significant cloud resources can now be executed locally on low-power microcontrollers.

The regional tier has also evolved beyond its traditional role of data aggregation. Modern regional nodes leverage techniques such as federated learning, where multiple devices collaboratively improve a shared model without transferring raw data to a central location. This approach not only enhances data privacy but also reduces bandwidth requirements. Regional tiers are increasingly used to adapt global models to local conditions, enabling more accurate and context-aware decision-making for specific deployment environments. This adaptability makes the regional tier an indispensable component for systems operating in diverse or resource-variable settings.

The relationship between the tiers has become more fluid and dynamic with these advancements. As edge and regional capabilities have expanded, the distribution of tasks across tiers is now determined by factors such as real-time resource availability, network conditions, and application requirements. For instance, during periods of low connectivity, edge and regional tiers can temporarily take on additional responsibilities to ensure critical functionality, while seamlessly offloading tasks to the cloud when resources and connectivity improve. This dynamic allocation preserves the hierarchical structure's inherent benefits, including scalability, resilience, and efficiency, while enabling greater adaptability to changing conditions.

These adaptations indicate future developments in Hierarchical Processing Pattern systems. As edge computing capabilities continue to advance and new distributed learning approaches emerge, the boundaries between tiers will likely become increasingly dynamic. This evolution suggests a future where hierarchical systems can automatically optimize their structure based on deployment context, resource availability, and application requirements, while maintaining the pattern's fundamental benefits of scalability, resilience, and efficiency.

19.6.1.4 System Implications

While the Hierarchical Processing Pattern was originally designed for general-purpose distributed systems, its application to machine learning introduces unique considerations that significantly influence system design and operation. Machine learning systems differ from traditional systems in their heavy reliance on data flows, computationally intensive tasks, and the dynamic nature of model updates and inference processes. These additional factors introduce both challenges and opportunities in adapting the Hierarchical Processing Pattern to meet the needs of machine learning deployments.

One of the most significant implications for machine learning is the need to manage dynamic model behavior across tiers. Unlike static systems, ML models require regular updates to adapt to new data distributions, prevent model drift, and maintain accuracy. The hierarchical structure inherently supports this requirement by allowing the cloud tier to handle centralized training and model updates while propagating refined models to regional and edge tiers. However, this introduces challenges in synchronization, as edge and regional tiers must continue operating with older model versions when updates are delayed due to connectivity issues. Designing robust versioning systems and ensuring seamless transitions between model updates is critical to the success of such systems.

Data flows are another area where machine learning systems impose unique demands. Unlike traditional hierarchical systems, ML systems must handle large volumes of data across tiers, ranging from raw inputs at the edge to aggregated and preprocessed datasets at regional and cloud tiers. Each tier must be optimized for the specific data-processing tasks it performs. For instance, edge devices often filter or preprocess raw data to reduce transmission overhead while retaining information critical for inference. Regional tiers aggregate these inputs, performing intermediate-level analysis or feature extraction to support downstream tasks. This multistage data pipeline not only reduces bandwidth requirements but also ensures that each tier contributes meaningfully to the overall ML workflow.

The Hierarchical Processing Pattern also enables adaptive inference, a key consideration for deploying ML models across environments with varying computational resources. By leveraging the computational capabilities of each tier, systems can dynamically distribute inference tasks to balance latency, energy consumption, and accuracy. For example, an edge device might handle basic anomaly detection to ensure real-time responses, while more sophisticated inference tasks are offloaded to the cloud when resources and connectivity allow. This dynamic distribution is essential for resource-constrained environments, where energy efficiency and responsiveness are paramount.

Hardware advancements have further shaped the application of the Hierarchical Processing Pattern to machine learning. The proliferation of specialized edge hardware, such as AI accelerators and low-power GPUs, has enabled edge devices to handle increasingly complex ML tasks, narrowing the performance gap between tiers. Regional tiers have similarly benefited from innovations such as federated learning, where models are collaboratively improved across devices without requiring centralized data collection. These advancements enhance the autonomy of lower tiers, reducing the dependency on cloud connectivity and enabling systems to function effectively in decentralized environments.

Finally, machine learning introduces the challenge of balancing local autonomy with global coordination. Edge and regional tiers must be able to make localized decisions based on the data available to them while remaining synchronized with the global state maintained at the cloud tier. This requires careful design of interfaces between tiers to manage not only data flows but also model updates, inference results, and feedback loops. For instance, systems employing federated learning must coordinate the aggregation of locally trained model updates without overwhelming the cloud tier or compromising privacy and security.

By integrating machine learning into the Hierarchical Processing Pattern, systems gain the ability to scale their capabilities across diverse environments, adapt dynamically to changing resource conditions, and balance real-time responsiveness with centralized intelligence. However, these benefits come with added complexity, requiring careful attention to model lifecycle management, data structuring, and resource allocation. The Hierarchical Processing Pattern remains a powerful framework for ML systems, enabling them to overcome the constraints of infrastructure variability while delivering high-impact solutions across a wide range of applications.

19.6.1.5 Limitations

Despite its strengths, the Hierarchical Processing Pattern encounters several fundamental constraints in real-world deployments, particularly when applied to machine learning systems. These limitations arise from the distributed nature of the architecture, the variability of resource availability across tiers, and the inherent complexities of maintaining consistency and efficiency at scale.

The distribution of processing capabilities introduces significant complexity in resource allocation and cost management. Regional processing nodes must navigate trade-offs between local computational needs, hardware costs, and energy consumption. In battery-powered deployments, the energy efficiency of local computation versus data transmission becomes a critical factor. These constraints directly affect the scalability and operational costs of the system, as additional nodes or tiers may require significant investment in infrastructure and hardware.

Time-critical operations present unique challenges in hierarchical systems. While edge processing reduces latency for local decisions, operations requiring cross-tier coordination introduce unavoidable delays. For instance, anomaly detection systems that require consensus across multiple regional nodes face inherent latency limitations. This coordination overhead can make hierarchical architectures unsuitable for applications requiring sub-millisecond response times or strict global consistency.

Training data imbalances across regions create additional complications. Different deployment environments often generate varying quantities and types of data, leading to model bias and performance disparities. For example, urban areas typically generate more training samples than rural regions, potentially causing models to underperform in less data-rich environments. This imbalance can be particularly problematic in systems where model performance directly impacts critical decision-making processes.

System maintenance and debugging introduce practical challenges that grow with scale. Identifying the root cause of performance degradation becomes increasingly complex when issues can arise from hardware failures, network conditions, model drift, or interactions between tiers. Traditional debugging approaches often prove inadequate, as problems may manifest only under specific combinations of conditions across multiple tiers. This complexity increases operational costs and requires specialized expertise for system maintenance.

These limitations necessitate careful consideration of mitigation strategies during system design. Approaches such as asynchronous processing protocols, tiered security frameworks, and automated debugging tools can help address specific challenges. Additionally, implementing robust monitoring systems that track performance metrics across tiers enables early detection of potential issues. While these limitations don't diminish the pattern's overall utility, they underscore the importance of thorough planning and risk assessment in hierarchical system deployments.

19.6.2 Progressive Enhancement

The progressive enhancement pattern applies a layered approach to system design, enabling functionality across environments with varying resource capacities. This pattern operates by establishing a baseline capability that remains operational under minimal resource conditions, typically requiring merely kilobytes of memory and milliwatts of power, and incrementally incorporating advanced features as additional resources become available. While originating from web development, where applications adapted to diverse browser capabilities and network conditions, the pattern has evolved to address the complexities of distributed systems and machine learning deployments.

This approach fundamentally differs from the Hierarchical Processing Pattern by focusing on vertical feature enhancement rather than horizontal distribution of tasks. Systems adopting this pattern are structured to maintain operations even under severe resource constraints, such as 2G network connections (< 50 kbps) or microcontroller-class devices (< 1 MB RAM). Additional capabilities are activated systematically as resources become available, with each enhancement layer building upon the foundation established by previous layers. This granular approach to resource utilization ensures system reliability while maximizing performance potential.

In machine learning applications, the progressive enhancement pattern enables sophisticated adaptation of models and workflows based on available resources. For instance, a computer vision system might deploy a 100 KB quantized model capable of basic object detection under minimal conditions, progressively expanding to more sophisticated models (1-50 MB) with higher accuracy and additional detection capabilities as computational resources permit. This adaptability allows systems to scale their capabilities dynamically while maintaining fundamental functionality across diverse operating environments.

19.6.2.1 PlantVillage Nuru

PlantVillage Nuru exemplifies the progressive enhancement pattern in its approach to providing AI-powered agricultural support for smallholder farmers (Ferentinos 2018), particularly in low-resource settings. Developed to address the challenges of crop diseases and pest management, Nuru combines machine learning models with mobile technology to deliver actionable insights directly to farmers, even in remote regions with limited connectivity or computational resources.[9]

PlantVillage Nuru operates with a baseline model optimized for resource-constrained environments. The system employs quantized convolutional neural

[9] PlantVillage Nuru has significantly impacted agricultural resilience, enabling farmers in over 60 countries to diagnose crop diseases with 85-90 percent accuracy using entry-level smartphones. The initiative has directly contributed to improved crop yields and reduced losses in vulnerable farming communities by integrating on-device AI and cloud-based insights.

networks (typically 2-5 MB in size) running on entry-level smartphones, capable of processing images at 1-2 frames per second while consuming less than 100mW of power. These on-device models achieve 85-90% accuracy in identifying common crop diseases, providing essential diagnostic capabilities without requiring network connectivity.

When network connectivity becomes available (even at 2G speeds of 50-100 kbps), Nuru progressively enhances its capabilities. The system uploads collected data to cloud infrastructure, where more sophisticated models (50-100 MB) perform advanced analysis with 95-98% accuracy. These models integrate multiple data sources: high-resolution satellite imagery (10-30 m resolution), local weather data (updated hourly), and soil sensor readings. This enhanced processing generates detailed mitigation strategies, including precise pesticide dosage recommendations and optimal timing for interventions.

In regions lacking widespread smartphone access, Nuru implements an intermediate enhancement layer through community digital hubs. These hubs, equipped with mid-range tablets (2 GB RAM, quad-core processors), cache diagnostic models and agricultural databases (10-20 GB) locally. This architecture enables offline access to enhanced capabilities while serving as data aggregation points when connectivity becomes available, typically synchronizing with cloud services during off-peak hours to optimize bandwidth usage.

This implementation demonstrates how progressive enhancement can scale from basic diagnostic capabilities to comprehensive agricultural support based on available resources. The system maintains functionality even under severe constraints (offline operation, basic hardware) while leveraging additional resources when available to provide increasingly sophisticated analysis and recommendations.

19.6.2.2 Structure

The progressive enhancement pattern organizes systems into layered functionalities, each designed to operate within specific resource conditions. This structure begins with a set of capabilities that function under minimal computational or connectivity constraints, progressively incorporating advanced features as additional resources become available.

Table 19.2 outlines the resource specifications and capabilities across the pattern's three primary layers:

Table 19.2: Resource specifications and capabilities across progressive enhancement pattern layers

Resource Type	Baseline Layer	Intermediate Layer	Advanced Layer
Computational	Microcontroller-class (100-200 MHz CPU, < 1MB RAM)	Entry-level smartphones (1-2 GB RAM)	Cloud/edge servers (8 GB+ RAM)
Network	Offline or 2G/GPRS	Intermittent 3G/4G (1-10 Mbps)	Reliable broadband (50 Mbps+)
Storage	Essential models (1-5 MB)	Local cache (10-50 MB)	Distributed systems (GB+ scale)
Power	Battery-operated (50-150 mW)	Daily charging cycles	Continuous grid power
Processing	Basic inference tasks	Moderate ML workloads	Full training capabilities
Data Access	Pre-packaged datasets	Periodic synchronization	Real-time data integration

Each layer in the progressive enhancement pattern operates independently, so that systems remain functional regardless of the availability of higher tiers. The pattern's modular structure enables seamless transitions between layers, minimizing disruptions as systems dynamically adjust to changing resource conditions. By prioritizing adaptability, the progressive enhancement pattern supports a wide range of deployment environments, from remote, resource-constrained regions to well-connected urban centers.

Figure 19.5 illustrates these three layers, showing the functionalities at each layer. The diagram visually demonstrates how each layer scales up based on available resources and how the system can fallback to lower layers when resource constraints occur.

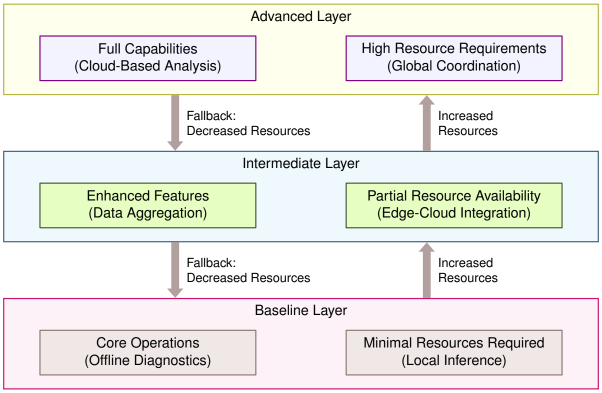

Advanced Layer
Full Capabilities (Cloud-Based Analysis)
High Resource Requirements (Global Coordination)
Fallback: Decreased Resources
Increased Resources
Intermediate Layer
Enhanced Features (Data Aggregation)
Partial Resource Availability (Edge-Cloud Integration)
Fallback: Decreased Resources
Increased Resources
Baseline Layer
Core Operations (Offline Diagnostics)
Minimal Resources Required (Local Inference)

Figure 19.5: Progressive enhancement pattern with specific examples of functionality at each layer.

19.6.2.3 Modern Adaptations

Modern implementations of the progressive enhancement pattern incorporate automated optimization techniques to create sophisticated resource-aware systems. These adaptations fundamentally reshape how systems manage varying resource constraints across deployment environments.

Automated architecture optimization represents a significant advancement in implementing progressive enhancement layers. Contemporary systems employ Neural Architecture Search to generate model families optimized for specific resource constraints. For example, a computer vision system might maintain multiple model variants ranging from 500 KB to 50 MB in size, each preserving maximum accuracy within its respective computational bounds. This automated approach ensures consistent performance scaling across enhancement layers, while setting the foundation for more sophisticated adaptation mechanisms.

Knowledge distillation and transfer mechanisms have evolved to support progressive capability enhancement. Modern systems implement bidirectional distillation processes where simplified models operating in resource-constrained environments gradually incorporate insights from their more sophisticated counterparts. This architectural approach enables baseline models to improve their performance over time while operating within strict resource limitations, creating a dynamic learning ecosystem across enhancement layers.

The evolution of distributed learning frameworks further extends these enhancement capabilities through federated optimization strategies. Base layer devices participate in simple model averaging operations, while better-resourced nodes implement more sophisticated federated optimization algorithms. This tiered approach to distributed learning enables system-wide improvements while respecting the computational constraints of individual devices, effectively scaling learning capabilities across diverse deployment environments.

These distributed capabilities culminate in resource-aware neural architectures that exemplify recent advances in dynamic adaptation. These systems modulate their computational graphs based on available resources, automatically adjusting model depth, width, and activation functions to match current hardware capabilities. Such dynamic adaptation enables smooth transitions between enhancement layers while maintaining optimal resource utilization, representing the current state of the art in progressive enhancement implementations.

19.6.2.4 System Implications

The application of the progressive enhancement pattern to machine learning systems introduces unique architectural considerations that extend beyond traditional progressive enhancement approaches. These implications fundamentally affect model deployment strategies, inference pipelines, and system optimization techniques.

Model architecture design requires careful consideration of computational-accuracy trade-offs across enhancement layers. At the baseline layer, models must operate within strict computational bounds (typically 100-500 KB model size) while maintaining acceptable accuracy thresholds (usually 85-90% of full model performance). Each enhancement layer then incrementally incorporates more sophisticated architectural components, such as additional model layers, attention mechanisms, or ensemble techniques, scaling computational requirements in tandem with available resources.

Training pipelines present distinct challenges in progressive enhancement implementations. Systems must maintain consistent performance metrics across different model variants while enabling smooth transitions between enhancement layers. This necessitates specialized training approaches such as progressive knowledge distillation, where simpler models learn to mimic the behavior of their more complex counterparts within their computational constraints. Training objectives must balance multiple factors: baseline model efficiency, enhancement layer accuracy, and cross-layer consistency.

Inference optimization becomes particularly critical in progressive enhancement scenarios. Systems must dynamically adapt their inference strategies based on available resources, implementing techniques such as adaptive batching, dynamic quantization, and selective layer activation. These optimizations ensure efficient resource utilization while maintaining real-time performance requirements across different enhancement layers.

Model synchronization and versioning introduce additional complexity in progressively enhanced ML systems. As models operate across different resource tiers, systems must maintain version compatibility and manage model updates without disrupting ongoing operations. This requires robust versioning protocols that track model lineage across enhancement layers while ensuring backward compatibility for baseline operations.

19.6.2.5 Limitations

While the progressive enhancement pattern offers significant advantages for ML system deployment, it introduces several technical challenges that impact implementation feasibility and system performance. These challenges particularly affect model management, resource optimization, and system reliability.

Model version proliferation presents a fundamental challenge. Each enhancement layer typically requires multiple model variants (often 3-5 per layer) to handle different resource scenarios, creating a combinatorial explosion in model management overhead. For example, a computer vision system supporting three enhancement layers might require up to 15 different model versions, each needing individual maintenance, testing, and validation. This complexity increases exponentially when supporting multiple tasks or domains.

Performance consistency across enhancement layers introduces significant technical hurdles. Models operating at the baseline layer (typically limited to 100-500 KB size) must maintain at least 85-90% of the accuracy achieved by advanced models while using only 1-5% of the computational resources. Achieving this efficiency-accuracy trade-off becomes increasingly difficult as task complexity increases. Systems often struggle to maintain consistent inference behavior when transitioning between layers, particularly when handling edge cases or out-of-distribution inputs.

Resource allocation optimization presents another critical limitation. Systems must continuously monitor and predict resource availability while managing the overhead of these monitoring systems themselves. The decision-making process for switching between enhancement layers introduces additional latency (typically 50-200 ms), which can impact real-time applications. This overhead becomes particularly problematic in environments with rapidly fluctuating resource availability.

Infrastructure dependencies create fundamental constraints on system capabilities. While baseline functionality operates within minimal requirements (50-150 mW power consumption, 2G network speeds), achieving full system potential requires substantial infrastructure improvements. The gap between baseline and enhanced capabilities often spans several orders of magnitude in computational requirements, creating significant disparities in system performance across deployment environments.

User experience continuity suffers from the inherent variability in system behavior across enhancement layers. Output quality and response times can vary significantly—from basic binary classifications at the baseline layer to detailed probabilistic predictions with confidence intervals at advanced layers. These variations can undermine user trust, particularly in critical applications where consistency is essential.

These limitations necessitate careful consideration during system design and deployment. Successful implementations require robust monitoring systems, graceful degradation mechanisms, and clear communication of system capabilities at each enhancement layer. While these challenges don't negate the pattern's utility, they emphasize the importance of thorough planning and realistic expectation setting in progressive enhancement deployments.

19.6.3 Distributed Knowledge

The Distributed Knowledge Pattern addresses the challenges of collective learning and inference across decentralized nodes, each operating with local data and computational constraints. Unlike hierarchical processing, where tiers have distinct roles, this pattern emphasizes peer-to-peer knowledge sharing and collaborative model improvement. Each node contributes to the network's collective intelligence while maintaining operational independence.

This pattern builds on established Mobile ML and Tiny ML techniques to enable autonomous local processing at each node. Devices implement quantized models (typically 1-5 MB) for initial inference, while employing techniques like federated learning for collaborative model improvement. Knowledge sharing occurs through various mechanisms: model parameter updates, derived features, or processed insights, depending on bandwidth and privacy constraints. This distributed approach enables the network to leverage collective experiences while respecting local resource limitations.

The pattern particularly excels in environments where traditional centralized learning faces significant barriers. By distributing both data collection and model training across nodes, systems can operate effectively even with intermittent connectivity (as low as 1-2 hours of network availability per day) or severe bandwidth constraints (50-100 KB/day per node). This resilience makes it especially valuable for social impact applications operating in infrastructure-limited environments.

The distributed approach fundamentally differs from progressive enhancement by focusing on horizontal knowledge sharing rather than vertical capability enhancement. Each node maintains similar baseline capabilities while contributing to and benefiting from the network's collective knowledge, creating a robust system that remains functional even when significant portions of the network are temporarily inaccessible.

19.6.3.1 Wildlife Insights

Wildlife Insights demonstrates the Distributed Knowledge Pattern's application in conservation through distributed camera trap networks. The system exemplifies how decentralized nodes can collectively build and share knowledge while operating under severe resource constraints in remote wilderness areas.

Each camera trap functions as an independent processing node, implementing sophisticated edge computing capabilities within strict power and computational limitations. These devices employ lightweight convolutional neural networks for species identification, alongside efficient activity detection models for motion analysis. Operating within power constraints of 50-100 mW, the devices utilize adaptive duty cycling[10] to maximize battery life while maintaining continuous monitoring capabilities. This local processing approach enables each node to independently analyze and filter captured imagery, reducing raw

[10] Adaptive Duty Cycling: A technique in power management that dynamically adjusts the system's operation time to extend battery life.

image data from several megabytes to compact insight vectors of just a few kilobytes.

The system's Distributed Knowledge Pattern sharing architecture enables effective collaboration between nodes despite connectivity limitations. Camera traps form local mesh networks[11] using low-power radio protocols, sharing processed insights rather than raw data. This peer-to-peer communication allows the network to maintain collective awareness of wildlife movements and potential threats across the monitored area. When one node detects significant activity, including the presence of an endangered species or indications of poaching, this information propagates through the network, enabling coordinated responses even in areas with no direct connectivity to central infrastructure.

When periodic connectivity becomes available through satellite or cellular links, nodes synchronize their accumulated knowledge with cloud infrastructure. This synchronization process carefully balances the need for data sharing with bandwidth limitations, employing differential updates and compression techniques. The cloud tier then applies more sophisticated analytical models to understand population dynamics and movement patterns across the entire monitored region.

The Wildlife Insights implementation demonstrates how Distributed Knowledge Pattern sharing can maintain system effectiveness even in challenging environments. By distributing both processing and decision-making capabilities across the network, the system ensures continuous monitoring and rapid response capabilities while operating within the severe constraints of remote wilderness deployments. This approach has proven particularly valuable for conservation efforts, enabling real-time wildlife monitoring and threat detection across vast areas that would be impractical to monitor through centralized systems[12].

19.6.3.2 Structure

The Distributed Knowledge Pattern comprises specific architectural components designed to enable decentralized data collection, processing, and knowledge sharing. The pattern defines three primary structural elements: autonomous nodes, communication networks, and aggregation mechanisms.

Figure 19.6 illustrates the key components and their interactions within the Distributed Knowledge Pattern. Individual nodes (rectangular shapes) operate autonomously while sharing insights through defined communication channels. The aggregation layer (diamond shape) combines distributed knowledge, which feeds into the analysis layer (oval shape) for processing.

Autonomous nodes form the foundation of the pattern's structure. Each node implements three essential capabilities: data acquisition, local processing, and knowledge sharing. The local processing pipeline typically includes feature extraction, basic inference, and data filtering mechanisms. This architecture enables nodes to operate independently while contributing to the network's collective intelligence.

The communication layer establishes pathways for knowledge exchange between nodes. This layer implements both peer-to-peer protocols for direct node communication and hierarchical protocols for aggregation. The communication

[11] Mesh Network: A network topology in which each node relays data for the network. All nodes cooperate in the distribution of data in the network.

[12] Camera traps have been widely used for ecological monitoring since the early 20th century. Initially reliant on physical film, they transitioned to digital and, more recently, AI-enabled systems, enhancing their ability to automate data analysis and extend deployment durations.

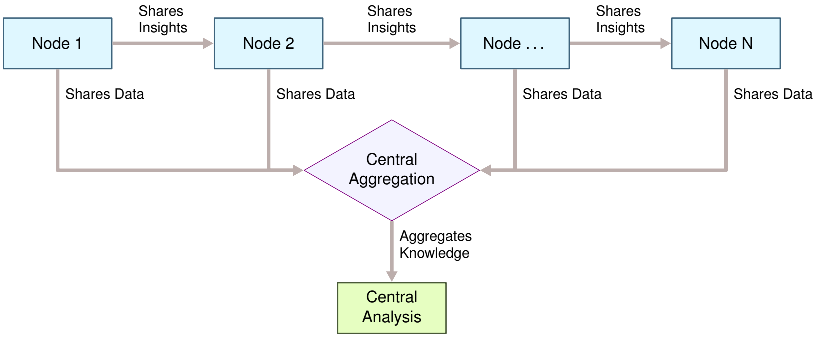


Figure 19.6: Distributed Knowledge Pattern with differentiated shapes for nodes, central aggregation, and analysis.

architecture must balance bandwidth efficiency with information completeness, often employing techniques such as differential updates and compressed knowledge sharing.

The aggregation and analysis layers provide mechanisms for combining distributed insights into understanding. These layers implement more sophisticated processing capabilities while maintaining feedback channels to individual nodes. Through these channels, refined models and updated processing parameters flow back to the distributed components, creating a continuous improvement cycle.

This structural organization ensures system resilience while enabling scalable knowledge sharing across distributed environments. The pattern's architecture specifically addresses the challenges of unreliable infrastructure and limited connectivity while maintaining system effectiveness through decentralized operations.

19.6.3.3 Modern Adaptations

The Distributed Knowledge Pattern has seen significant advancements with the rise of modern technologies like edge computing, the Internet of Things (IoT), and decentralized data networks. These innovations have enhanced the scalability, efficiency, and flexibility of systems utilizing this pattern, enabling them to handle increasingly complex data sets and to operate in more diverse and challenging environments.

One key adaptation has been the use of edge computing. Traditionally, distributed systems rely on transmitting data to centralized servers for analysis. However, with edge computing, nodes can perform more complex processing locally, reducing the dependency on central systems and enabling real-time data processing. This adaptation has been especially impactful in areas where network connectivity is intermittent or unreliable. For example, in remote wildlife conservation systems, camera traps can process images locally and only transmit relevant insights, such as the detection of a poacher, to a central hub when connectivity is restored. This reduces the amount of raw data sent across the network and ensures that the system remains operational even in areas with limited infrastructure.

Another important development is the integration of machine learning at the edge. In traditional distributed systems, machine learning models are often centralized, requiring large amounts of data to be sent to the cloud for processing. With the advent of smaller, more efficient machine learning models designed for edge devices, these models can now be deployed directly on the nodes themselves. For example, low-power devices such as smartphones or IoT sensors can run lightweight models for tasks like anomaly detection or image classification. This enables more sophisticated data analysis at the source, allowing for quicker decision-making and reducing reliance on central cloud services.

In terms of network communication, modern mesh networks and 5G technology have significantly improved the efficiency and speed of data sharing between nodes. Mesh networks allow nodes to communicate with each other directly, forming a self-healing and scalable network. This decentralized approach to communication ensures that even if a node or connection fails, the network can still operate seamlessly. With the advent of 5G, the bandwidth and latency issues traditionally associated with large-scale data transfer in distributed systems are mitigated, enabling faster and more reliable communication between nodes in real-time applications.

19.6.3.4 System Implications

The Distributed Knowledge Pattern fundamentally reshapes how machine learning systems handle data collection, model training, and inference across decentralized nodes. These implications extend beyond traditional distributed computing challenges to encompass ML-specific considerations in model architecture, training dynamics, and inference optimization.

Model architecture design requires specific adaptations for distributed deployment. Models must be structured to operate effectively within node-level resource constraints while maintaining sufficient complexity for accurate inference. This often necessitates specialized architectures that support incremental learning and knowledge distillation. For instance, neural network architectures might implement modular components that can be selectively activated based on local computational resources, typically operating within 1-5MB memory constraints while maintaining 85-90% of centralized model accuracy.

Training dynamics become particularly complex in Distributed Knowledge Pattern systems. Unlike centralized training approaches, these systems must implement collaborative learning mechanisms that function effectively across unreliable networks. Federated averaging protocols must be adapted to handle non-IID (Independent and Identically Distributed) data distributions across nodes while maintaining convergence guarantees. Training procedures must also account for varying data qualities and quantities across nodes, implementing weighted aggregation schemes that reflect data reliability and relevance.

Inference optimization presents unique challenges in distributed environments. Models must adapt their inference strategies based on local resource availability while maintaining consistent output quality across the network. This often requires implementing dynamic batching strategies, adaptive quantization, and selective feature computation. Systems typically target sub-100 ms inference latency at the node level while operating within strict power envelopes (50-150 mW).

Model lifecycle management becomes significantly more complex in Distributed Knowledge Pattern systems. Version control must handle multiple model variants operating across different nodes, managing both forward and backward compatibility. Systems must implement robust update mechanisms that can handle partial network connectivity while preventing model divergence across the network.

19.6.3.5 Limitations

While the Distributed Knowledge Pattern offers many advantages, particularly in decentralized, resource-constrained environments, it also presents several challenges, especially when applied to machine learning systems. These challenges stem from the complexity of managing distributed nodes, ensuring data consistency, and addressing the constraints of decentralized systems.

One of the primary challenges is model synchronization and consistency. In distributed systems, each node may operate with its own version of a machine learning model, which is trained using local data. As these models are updated over time, ensuring consistency across all nodes becomes a difficult task. Without careful synchronization, nodes may operate using outdated models, leading to inconsistencies in the system's overall performance. Furthermore, when nodes are intermittently connected or have limited bandwidth, synchronizing model updates across all nodes in real-time can be resource-intensive and prone to delays.

The issue of data fragmentation is another significant challenge. In a distributed system, data is often scattered across different nodes, and each node may have access to only a subset of the entire dataset. This fragmentation can limit the effectiveness of machine learning models, as the models may not be exposed to the full range of data needed for training. Aggregating data from multiple sources and ensuring that the data from different nodes is compatible for analysis is a complex and time-consuming process. Additionally, because some nodes may operate in offline modes or have intermittent connectivity, data may be unavailable for periods, further complicating the process.

Scalability also poses a challenge in distributed systems. As the number of nodes in the network increases, so does the volume of data generated and the complexity of managing the system. The system must be designed to handle this growth without overwhelming the infrastructure or degrading performance. The addition of new nodes often requires rebalancing data, recalibrating models, or introducing new coordination mechanisms, all of which can increase the complexity of the system.

Latency is another issue that arises in distributed systems. While data is processed locally on each node, real-time decision-making often requires the aggregation of insights from multiple nodes. The time it takes to share data and updates between nodes, and the time needed to process that data, can introduce delays in system responsiveness. In applications like autonomous systems or disaster response, these delays can undermine the effectiveness of the system, as immediate action is often necessary.

Finally, security and privacy concerns are magnified in distributed systems. Since data is often transmitted between nodes or stored across multiple devices, ensuring the integrity and confidentiality of the data becomes a significant challenge. The system must employ strong encryption and authentication mechanisms to prevent unauthorized access or tampering of sensitive information. This is especially important in applications involving private or protected data, such as healthcare or financial systems. Additionally, decentralized systems may be more susceptible to certain types of attacks, such as Sybil attacks, where an adversary can introduce fake nodes into the network.

Despite these challenges, there are several strategies that can help mitigate the limitations of the Distributed Knowledge Pattern. For example, federated learning techniques can help address model synchronization issues by enabling nodes to update models locally and only share the updates, rather than raw data. Decentralized data aggregation methods can help address data fragmentation by allowing nodes to perform more localized aggregation before sending data to higher tiers. Similarly, edge computing can reduce latency by processing data closer to the source, reducing the time needed to transmit information to central servers.

19.6.4 Adaptive Resource

The Adaptive Resource Pattern focuses on enabling systems to dynamically adjust their operations in response to varying resource availability, ensuring efficiency, scalability, and resilience in real-time. This pattern allows systems to allocate resources flexibly depending on factors like computational load, network bandwidth, and storage capacity. The key idea is that systems should be able to scale up or down based on the resources they have access to at any given time.

Rather than being a standalone pattern, Adaptive Resource Pattern management is often integrated within other system design patterns. It enhances systems by allowing them to perform efficiently even under changing conditions, ensuring that they continue to meet their objectives, regardless of resource fluctuations.

Figure 19.7 below illustrates how systems using the Adaptive Resource Pattern adapt to different levels of resource availability. The system adjusts its operations based on the resources available at the time, optimizing its performance accordingly.

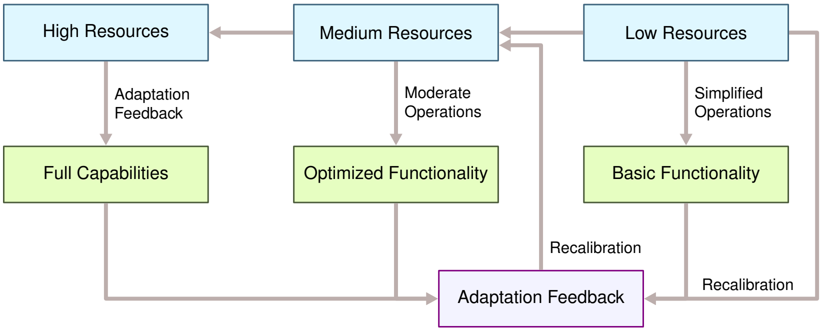


Figure 19.7: The Adaptive Resource Pattern.

In the diagram, when the system is operating under low resources, it switches to simplified operations, ensuring basic functionality with minimal resource use. As resources become more available, the system adjusts to medium resources, enabling more moderate operations and optimized functionality. When resources are abundant, the system can leverage high resources, enabling advanced operations and full capabilities, such as processing complex data or running resource-intensive tasks.

The feedback loop is an essential part of this pattern, as it ensures continuous adjustment based on the system's resource conditions. This feedback allows the system to recalibrate and adapt in real-time, scaling resources up or down to maintain optimal performance.

19.6.4.1 Case Studies

Looking at the systems we discussed earlier, it is clear that these systems could benefit from Adaptive Resource Pattern allocation in their operations. In the case of Google's flood forecasting system, the Hierarchical Processing Pattern approach ensures that data is processed at the appropriate level, from edge sensors to cloud-based analysis. However, Adaptive Resource Pattern management would enable this system to adjust its operations dynamically depending on the resources available. In areas with limited infrastructure, the system could rely more heavily on edge processing to reduce the need for constant connectivity, while in regions with better infrastructure, the system could scale up and leverage more cloud-based processing power.

Similarly, PlantVillage Nuru could integrate Adaptive Resource Pattern allocation into its progressive enhancement approach. The app is designed to work in a variety of settings, from low-resource rural areas to more developed regions. The Adaptive Resource Pattern management in this context would help the system adjust the complexity of its processing based on the available device and network resources, ensuring that it provides useful insights without overwhelming the system or device.

In the case of Wildlife Insights, the Adaptive Resource Pattern management would complement the Distributed Knowledge Pattern. The camera traps in the field process data locally, but when network conditions improve, the system could scale up to transmit more data to central systems for deeper analysis. By using adaptive techniques, the system ensures that the camera traps can continue to function even with limited power and network connectivity, while still providing valuable insights when resources allow for greater computational effort.

These systems could integrate the Adaptive Resource Pattern management to dynamically adjust based on available resources, improving efficiency and ensuring continuous operation under varying conditions. By incorporating the Adaptive Resource Pattern allocation into their design, these systems can remain responsive and scalable, even as resource availability fluctuates. The Adaptive Resource Pattern, in this context, acts as an enabler, supporting the operations of these systems and helping them adapt to the demands of real-time environments.

19.6.4.2 Structure

The Adaptive Resource Pattern revolves around dynamically allocating resources in response to changing environmental conditions, such as network bandwidth, computational power, or storage. This requires the system to monitor available resources continuously and adjust its operations accordingly to ensure optimal performance and efficiency.

It is structured around several key components. First, the system needs a monitoring mechanism to constantly evaluate the availability of resources. This can involve checking network bandwidth, CPU utilization, memory usage, or other relevant metrics. Once these metrics are gathered, the system can then determine the appropriate course of action—whether it needs to scale up, down, or adjust its operations to conserve resources.

Next, the system must include an adaptive decision-making process that interprets these metrics and decides how to allocate resources dynamically. In high-resource environments, the system might increase the complexity of tasks, using more powerful computational models or increasing the number of concurrent processes. Conversely, in low-resource environments, the system may scale back operations, reduce the complexity of models, or shift some tasks to local devices (such as edge processing) to minimize the load on the central infrastructure.

An important part of this structure is the feedback loop, which allows the system to adjust its resource allocation over time. After making an initial decision based on available resources, the system monitors the outcome and adapts accordingly. This process ensures that the system continues to operate effectively even as resource conditions change. The feedback loop helps the system fine-tune its resource usage, leading to more efficient operations as it learns to optimize resource allocation.

The system can also be organized into different tiers or layers based on the complexity and resource requirements of specific tasks. For instance, tasks requiring high computational resources, such as training machine learning models or processing large datasets, could be handled by a cloud layer, while simpler tasks, such as data collection or pre-processing, could be delegated to edge devices or local nodes. The system can then adapt the tiered structure based on available resources, allocating more tasks to the cloud or edge depending on the current conditions.

19.6.4.3 Modern Adaptations

The Adaptive Resource Pattern has evolved significantly with advancements in cloud computing, edge computing, and AI-driven resource management. These innovations have enhanced the flexibility and scalability of the pattern, allowing it to adapt more efficiently in increasingly complex environments.

One of the most notable modern adaptations is the integration of cloud computing. Cloud platforms like AWS, Microsoft Azure, and Google Cloud offer the ability to dynamically allocate resources based on demand, making it easier to scale applications in real-time. This integration allows systems to offload intensive processing tasks to the cloud when resources are available and return to more efficient, localized solutions when demand decreases or resources are constrained. The elasticity provided by cloud computing enables systems to perform heavy computational tasks, such as machine learning model training or big data processing, without requiring on-premise infrastructure.

At the other end of the spectrum, edge computing has emerged as a critical adaptation for the Adaptive Resource Pattern. In edge computing, data is processed locally on devices or at the edge of the network, reducing the dependency on centralized servers and improving real-time responsiveness. Edge devices, such as IoT sensors or smartphones, often operate in resource-constrained environments, and the ability to process data locally allows for more efficient use of limited resources. By offloading certain tasks to the edge, systems can maintain functionality even in low-resource areas while ensuring that computationally intensive tasks are shifted to the cloud when available.

The rise of AI-driven resource management has also transformed how adaptive systems function. AI can now monitor resource usage patterns in real-time and predict future resource needs, allowing systems to adjust resource allocation proactively. For example, machine learning models can be trained to identify patterns in network traffic, processing power, or storage utilization, enabling the system to predict peak usage times and prepare resources accordingly. This proactive adaptation ensures that the system can handle fluctuations in demand smoothly and without interruption, reducing latency and improving overall system performance.

These modern adaptations allow systems to perform complex tasks while adapting to local conditions. For example, in disaster response systems, resources such as rescue teams, medical supplies, and communication tools can be dynamically allocated based on the evolving needs of the situation. Cloud computing enables large-scale coordination, while edge computing ensures that critical decisions can be made at the local level, even when the network is down. By integrating AI-driven resource management, the system can predict resource shortages or surpluses, ensuring that resources are allocated in the most effective way.

These modern adaptations make the Adaptive Resource Pattern more powerful and flexible than ever. By leveraging cloud, edge computing, and AI, systems can dynamically allocate resources across distributed environments, ensuring that they remain scalable, efficient, and resilient in the face of changing conditions.

19.6.4.4 System Implications

Adaptive Resource Pattern has significant implications for machine learning systems, especially when deployed in environments with fluctuating resources, such as mobile devices, edge computing platforms, and distributed systems. Machine learning workloads can be resource-intensive, requiring substantial computational power, memory, and storage. By integrating the Adaptive Resource Pattern allocation, ML systems can optimize their performance, ensure scalability, and maintain efficiency under varying resource conditions.

In the context of distributed machine learning (e.g., federated learning), the Adaptive Resource Pattern ensures that the system adapts to varying computational capacities across devices. For example, in federated learning, models are trained collaboratively across many edge devices (such as smartphones or IoT devices), where each device has limited resources. The Adaptive Resource Pattern management can allocate the model training tasks based on the resources available on each device. Devices with more computational power can handle heavier workloads, while devices with limited resources can participate in lighter tasks, such as local model updates or simple computations. This ensures that all devices can contribute to the learning process without overloading them.

Another implication of the Adaptive Resource Pattern in ML systems is its ability to optimize real-time inference. In applications like autonomous vehicles, healthcare diagnostics, and environmental monitoring, ML models need to make real-time decisions based on available data. The system must dynamically adjust its computational requirements based on the resources available at the time. For instance, an autonomous vehicle running an image recognition model may process simpler, less detailed frames when computing resources are constrained or when the vehicle is in a resource-limited area (e.g., an area with poor connectivity). When computational resources are more plentiful, such as in a connected city with high-speed internet, the system can process more detailed frames and apply more complex models.

The adaptive scaling of ML models also plays a significant role in cloud-based ML systems. In cloud environments, the Adaptive Resource Pattern allows the system to scale the number of resources used for tasks like model training or batch inference. When large-scale data processing or model training is required, cloud services can dynamically allocate resources to handle the increased load. When demand decreases, resources are scaled back to reduce operational costs. This dynamic scaling ensures that ML systems run efficiently and cost-effectively, without over-provisioning or underutilizing resources.

Additionally, AI-driven resource management is becoming an increasingly important component of adaptive ML systems. AI techniques, such as reinforcement learning or predictive modeling, can be used to optimize resource allocation in real-time. For example, reinforcement learning algorithms can be applied to predict future resource needs based on historical usage patterns, allowing systems to preemptively allocate resources before demand spikes. This proactive approach ensures that ML models are trained and inference tasks are executed with minimal latency, even as resources fluctuate.

Lastly, edge AI systems benefit greatly from the Adaptive Resource Pattern. These systems often operate in environments with highly variable resources, such as remote areas, rural regions, or environments with intermittent connectivity. The pattern allows these systems to adapt their resource allocation based on the available resources in real-time, ensuring that essential tasks, such as model inference or local data processing, can continue even in challenging conditions. For example, an environmental monitoring system deployed in a remote area may adapt by running simpler models or processing less detailed data when resources are low, while more complex analysis is offloaded to the cloud when the network is available.

19.6.4.5 Limitations

The Adaptive Resource Pattern faces several fundamental constraints in practical implementations, particularly when applied to machine learning systems in resource-variable environments. These limitations arise from the inherent complexities of real-time adaptation and the technical challenges of maintaining system performance across varying resource levels.

Performance predictability presents a primary challenge in adaptive systems. While adaptation enables systems to continue functioning under varying conditions, it can lead to inconsistent performance characteristics. For example, when a system transitions from high to low resource availability (e.g., from 8 GB to 500 MB RAM), inference latency might increase from 50 ms to 200 ms. Managing these performance variations while maintaining minimum quality-of-service requirements becomes increasingly complex as the range of potential resource states expands.

State synchronization introduces significant technical hurdles in adaptive systems. As resources fluctuate, maintaining consistent system state across components becomes challenging. For instance, when adapting to reduced network bandwidth (from 50 Mbps to 50 Kbps), systems must manage partial updates and ensure that critical state information remains synchronized. This challenge is particularly acute in distributed ML systems, where model states and inference results must remain consistent despite varying resource conditions.

Resource transition overhead poses another fundamental limitation. Adapting to changing resource conditions incurs computational and time costs. For example, switching between different model architectures (from a 50 MB full model to a 5 MB quantized version) typically requires 100-200 ms of transition time. During these transitions, system performance may temporarily degrade or become unpredictable. This overhead becomes particularly problematic in environments where resources fluctuate frequently.

Quality degradation management presents ongoing challenges, especially in ML applications. As systems adapt to reduced resources, maintaining acceptable quality metrics becomes increasingly difficult. For instance, model accuracy might drop from 95% to 85% when switching to lightweight architectures, while energy consumption must stay within strict limits (typically 50-150 mW for edge devices). Finding acceptable trade-offs between resource usage and output quality requires sophisticated optimization strategies.

These limitations necessitate careful system design and implementation strategies. Successful deployments often implement robust monitoring systems, graceful degradation mechanisms, and clear quality thresholds for different resource states. While these challenges don't negate the pattern's utility, they emphasize the importance of thorough planning and realistic performance expectations in adaptive system deployments.

19.7 Selection Framework

The selection of an appropriate design pattern for machine learning systems in social impact contexts requires careful consideration of both technical constraints and operational requirements. Rather than treating patterns as rigid templates, system architects should view them as adaptable frameworks that can be tailored to specific deployment scenarios.

The selection process begins with a systematic analysis of four critical dimensions: resource variability, operational scale, data distribution requirements, and adaptation needs. Resource variability encompasses both the range and predictability of available computational resources, typically spanning from severely constrained environments (50-150 mW power, < 1 MB RAM) to resource-rich deployments (multi-core servers, GB+ RAM). Operational scale considers both geographic distribution and user base size, ranging from localized deployments to systems spanning multiple regions. Data distribution requirements address how information needs to flow through the system, from centralized architectures to fully distributed networks. Adaptation needs examine how dynamically the system must respond to changing conditions, from relatively stable environments to highly variable scenarios.

19.7.1 Selection Dimensions

These dimensions can be visualized through a quadrant analysis framework that maps patterns based on their resource requirements and adaptability needs. This approach simplifies understanding (at least from a pedagogical perspective) by providing a structured view of how systems align with varying constraints.

Figure 19.8 provides a structured approach for pattern selection based on two key axes: resource availability and scalability/adaptability needs. The horizontal axis corresponds to the level of computational, network, and power resources available to the system. Systems designed for resource-constrained environments, such as rural or remote areas, are positioned towards the left, while those leveraging robust infrastructure, such as cloud-supported systems, are placed towards the right. The vertical axis captures the system's ability to function across diverse settings or respond dynamically to changing conditions.

In low-resource environments with high adaptability needs, the progressive enhancement pattern dominates. Projects like PlantVillage Nuru implement Tiny ML and Mobile ML paradigms for offline crop diagnostics on basic smartphones. Similarly, Medic Mobile leverages these paradigms to support community health workers, enabling offline data collection and basic diagnostics that sync when connectivity permits.

For environments with higher resource availability and significant scalability demands, the Hierarchical Processing Pattern prevails. Google's Flood Forecasting Initiative exemplifies this approach, combining Edge ML for local sensor processing with Cloud ML for analytics. Global Fishing Watch similarly leverages this pattern, processing satellite data through a hierarchy of computational tiers to monitor fishing activities worldwide.

The Distributed Knowledge Pattern excels in low-resource environments requiring decentralized operations. Wildlife Insights demonstrates this through AI-enabled camera traps that employ Edge ML for local image processing while sharing insights across peer networks. WildEyes AI follows a similar approach, using distributed nodes for poaching detection with minimal central coordination.

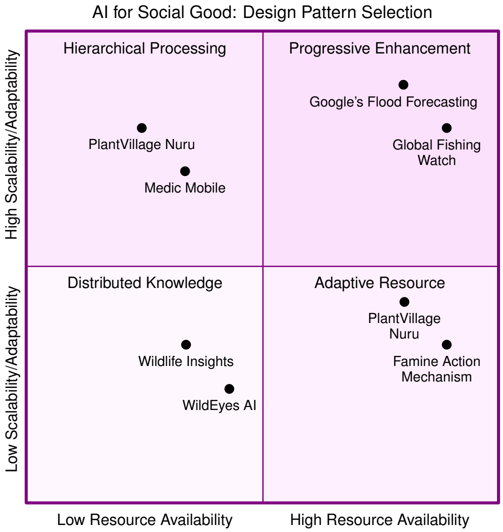


Figure 19.8: Quadrant mapping of design patterns for AI for Social Good projects based on resource availability and scalability/adaptability needs.

Systems requiring dynamic resource allocation in fluctuating environments benefit from the Adaptive Resource Pattern. AI for Disaster Response exemplifies this approach, combining Edge ML for immediate local processing with Cloud ML scalability during crises. The AI-powered Famine Action Mechanism similarly adapts its resource allocation dynamically, scaling analysis capabilities based on emerging conditions and available infrastructure.

19.7.2 Implementation Guidance

As outlined in Table 19.3, each pattern presents distinct strengths and challenges that influence implementation decisions. The practical deployment of these patterns requires careful consideration of both the operational context and the specific requirements of machine learning systems.

Table 19.3: Comparisons of design patterns.

Design Pattern	Core Idea	Strengths	Challenges	Best Use Case
Hierarchical Processing	Organizes operations into edge, regional, and cloud tiers.	Scalability, resilience, fault tolerance	Synchronization issues, model versioning, and latency in updates.	Distributed workloads spanning diverse infrastructures (e.g., Google's Flood Forecasting).
Progressive Enhancement	Provides baseline functionality and scales up dynamically.	Adaptability to resource variability, inclusivity	Ensuring consistent UX and increased complexity in layered design.	Applications serving both resource-constrained and resource-rich environments (e.g., PlantVillage Nuru).
Distributed Knowledge	Decentralizes data processing and sharing across nodes.	Resilient in low-bandwidth environments, scalability	Data fragmentation and challenges with synchronizing decentralized models.	Systems requiring collaborative, decentralized insights (e.g., Wildlife Insights for conservation).

Design Pattern	Core Idea	Strengths	Challenges	Best Use Case
Adaptive Resource	Dynamically adjusts operations based on resource availability.	Resource efficiency and real-time adaptability	Predicting resource demand and managing trade-offs between performance and simplicity.	Real-time systems operating under fluctuating resource conditions (e.g., disaster response systems).

The implementation approach for each pattern should align with both its position in the resource-adaptability space and its core characteristics. In low-resource, high-adaptability environments, Progressive Enhancement implementations focus on establishing reliable baseline capabilities that can scale smoothly as resources become available. This often involves careful coordination between local processing and cloud resources, ensuring that systems maintain functionality even when operating at minimal resource levels.

Hierarchical Processing Pattern implementations, suited for environments with more stable infrastructure, require careful attention to the interfaces between tiers. The key challenge lies in managing the flow of data and model updates across the hierarchy while maintaining system responsiveness. This becomes particularly critical in social impact applications where real-time response capabilities often determine intervention effectiveness.

Distributed Knowledge Pattern implementations emphasize resilient peer-to-peer operations, particularly important in environments where centralized coordination isn't feasible. Success depends on establishing efficient knowledge-sharing protocols that maintain system effectiveness while operating within strict resource constraints. This pattern's implementation often requires careful balance between local autonomy and network-wide consistency.

The Adaptive Resource Pattern implementations focus on dynamic resource management, particularly crucial in environments with fluctuating resource availability. These systems require sophisticated monitoring and control mechanisms that can adjust operations in real-time while maintaining essential functionality. The implementation challenge lies in managing these transitions smoothly without disrupting critical operations.

19.7.3 Comparison Analysis

Each design pattern offers unique advantages and trade-offs in ML system implementations. Understanding these distinctions enables system architects to make informed decisions based on deployment requirements and operational constraints.

The Hierarchical Processing Pattern and progressive enhancement pattern represent fundamentally different approaches to resource management. While the Hierarchical Processing Pattern establishes fixed infrastructure tiers with clear boundaries and responsibilities, progressive enhancement implements a continuous spectrum of capabilities that can scale smoothly with available resources. This distinction makes the Hierarchical Processing Pattern more suitable for environments with well-defined infrastructure tiers, while progressive enhancement better serves deployments where resource availability varies unpredictably.

The Distributed Knowledge Pattern and Adaptive Resource Pattern address different aspects of system flexibility. The Distributed Knowledge Pattern focuses on spatial distribution and peer-to-peer collaboration, while the Adaptive Resource Pattern management emphasizes temporal adaptation to changing conditions. These patterns can be complementary. The Distributed Knowledge Pattern handles geographic scale, while the Adaptive Resource Pattern management handles temporal variations in resource availability.

Selection between patterns often depends on the primary constraint facing the deployment. Systems primarily constrained by network reliability typically benefit from the Distributed Knowledge Pattern or Hierarchical Processing Pattern approaches. Those facing computational resource variability align better with progressive enhancement or Adaptive Resource Pattern approaches. The resource adaptability analysis presented earlier provides a structured framework for navigating these decisions based on specific deployment contexts.

19.8 Conclusion

The potential of AI for addressing societal challenges is undeniable. However, the path to successful deployment is anything but straightforward. ML systems for social good are not "plug-and-play" solutions, as they are complex engineering endeavors.

These systems must be tailored to operate under severe constraints, such as limited power, unreliable connectivity, and sparse data, all while meeting the needs of underserved communities. Designing for these environments is as rigorous and demanding as developing systems for urban deployments, often requiring even more ingenuity to overcome unique challenges. Every componen, from data collection to model deployment, must be reimagined to suit these constraints and deliver meaningful outcomes.

Machine learning systems for social impact necessitate the systematic application of design patterns to address these unique complexities. The patterns examined in this chapter, including Hierarchical Processing, Progressive Enhancement, Distributed Knowledge, and Adaptive Resource, establish frameworks for addressing these challenges while ensuring systems remain effective and sustainable across diverse deployment contexts.

The implementation of these patterns depends fundamentally on a comprehensive understanding of both the operational environment and system requirements. Resource availability and adaptability requirements typically determine initial pattern selection, while specific implementation decisions must account for network reliability, computational constraints, and scalability requirements. The efficacy of social impact applications depends not only on pattern selection but on implementation strategies that address local constraints while maintaining system performance.

These patterns will evolve as technological capabilities advance and deployment contexts transform. Developments in edge computing, federated learning, and adaptive ML architectures will expand the potential applications of these patterns, particularly in resource-constrained environments. However, the core principles, such as accessibility, reliability, and scalability, remain fundamental to developing ML systems that generate meaningful social impact.

The systematic application of these design patterns, informed by rigorous analysis of deployment contexts and constraints, enables the development of ML systems that function effectively across the computing spectrum while delivering sustainable social impact.

19.9 Resources

ℹ Slides

- *Coming soon.*

! Videos

- *Coming soon.*

🔥 Exercises

- *Coming soon.*

Chapter 20

Conclusion

Figure 20.1: *DALL·E 3 Prompt: An image depicting the last chapter of an ML systems book, open to a two-page spread. The pages summarize key concepts such as neural networks, model architectures, hardware acceleration, and MLOps. One page features a diagram of a neural network and different model architectures, while the other page shows illustrations of hardware components for acceleration and MLOps workflows. The background includes subtle elements like circuit patterns and data points to reinforce the technological theme. The colors are professional and clean, with an emphasis on clarity and understanding.*

20.1 Overview

This book examines the rapidly evolving field of ML systems. We focused on systems because while there are many resources on ML models and algorithms, more needs to be understood about how to build the systems that run them.

To draw an analogy, consider the process of building a car. While many resources are available on the various components of a car, such as the engine, transmission, and suspension, there is often a need for more understanding about how to assemble these components into a functional vehicle. Just as a car requires a well-designed and properly integrated system to operate efficiently and reliably, ML models also require a robust and carefully constructed system to deliver their full potential. Moreover, there is a lot of nuance in building ML systems, given their specific use case. For example, a Formula 1 race car must be assembled differently from an everyday Prius consumer car.

Our journey started by tracing ML's historical trajectory, from its theoretical foundations to its current state as a transformative force across industries. We explored the building blocks of machine learning models and demonstrated how their architectures, when examined through the lens of computer architecture, reveal structural similarities.

Throughout this book, we have looked into the intricacies of ML systems, examining the critical components and best practices necessary to create a seamless and efficient pipeline. From data preprocessing and model training to deployment and monitoring, we have provided insights and guidance to help readers navigate the complex landscape of ML system development.

ML systems involve complex workflows, spanning various topics from data engineering to model deployment on diverse systems. By providing an overview of these ML system components, we have aimed to showcase the tremendous depth and breadth of the field and expertise that is needed. Understanding the intricacies of ML workflows is crucial for practitioners and researchers alike, as it enables them to navigate the landscape effectively and develop robust, efficient, and impactful ML solutions.

By focusing on the systems aspect of ML, we aim to bridge the gap between theoretical knowledge and practical implementation. Just as a healthy human body system allows the organs to function optimally, a well-designed ML system enables the models to consistently deliver accurate and reliable results. This book's goal is to empower readers with the knowledge and tools necessary to build ML systems that showcase the underlying models' power and ensure smooth integration and operation, much like a well-functioning human body.

20.2 ML Dataset Importance

One of the key principles we have emphasized is that data is the foundation upon which ML systems are built. Data is the new code that programs deep neural networks, making data engineering the first and most critical stage of any ML pipeline. That is why we began our exploration by diving into the basics of data engineering, recognizing that quality, diversity, and ethical sourcing are key to building robust and reliable machine learning models.

The importance of high-quality data must be balanced. Lapses in data quality can lead to significant negative consequences, such as flawed predictions, project terminations, and even potential harm to communities. These cascading effects, highlight the need for diligent data management and governance practices. ML practitioners must prioritize data quality, ensure diversity and representativeness, and adhere to ethical data collection and usage standards. By doing so, we can mitigate the risks associated with poor data quality and build ML systems that are trustworthy, reliable, and beneficial to society.

20.3 AI Framework Navigation

Throughout this book, we have seen how machine learning frameworks serve as the backbone of modern ML systems. We dove into the evolution of different ML frameworks, dissecting the inner workings of popular ones like TensorFlow and PyTorch, and provided insights into the core components and advanced features that define them. We also looked into the specialization of frameworks tailored to specific needs, such as those designed for embedded AI. We discussed the criteria for selecting the most suitable framework for a given project.

Our exploration also touched upon the future trends expected to shape the landscape of ML frameworks in the coming years. As the field continues to evolve, we can anticipate the emergence of more specialized and optimized frameworks that cater to the unique requirements of different domains and deployment scenarios, as we saw with TensorFlow Lite for Microcontrollers. By staying abreast of these developments and understanding the tradeoffs involved in framework selection, we can make informed decisions and leverage the most appropriate tools to build efficient ML systems.

20.4 ML Training Basics

We saw how the AI training process is computationally intensive, making it challenging to scale and optimize. We began by examining the fundamentals of AI training, which involves feeding data into ML models and adjusting their parameters to minimize the difference between predicted and actual outputs. This process requires careful consideration of various factors, such as the choice of optimization algorithms, learning rate, batch size, and regularization techniques.

However, training ML models at scale poses significant system challenges. As datasets' size and models' complexity grow, the computational resources required for training can become prohibitively expensive. This has led to the development of distributed training techniques, such as data and model parallelism, which allow multiple devices to collaborate in the training process. Frameworks like TensorFlow and PyTorch have evolved to support these distributed training paradigms, enabling practitioners to scale their training workloads across clusters of GPUs or TPUs.

In addition to distributed training, we discussed techniques for optimizing the training process, such as mixed-precision training and gradient compression. It's important to note that while these techniques may seem algorithmic, they significantly impact system performance. The choice of training algorithms, precision, and communication strategies directly affects the ML system's resource utilization, scalability, and efficiency. Therefore, adopting an algorithm-hardware or algorithm-system co-design approach is crucial, where the algorithmic choices are made in tandem with the system considerations. By understanding the interplay between algorithms and hardware, we can make informed decisions that optimize the model performance and the system efficiency, ultimately leading to more effective and scalable ML solutions.

20.5 AI System Efficiency

Deploying trained ML models is more complex than simply running the networks; efficiency is critical. In this chapter on AI efficiency, we emphasized that efficiency is not merely a luxury but a necessity in artificial intelligence systems. We dug into the key concepts underpinning AI systems' efficiency, recognizing that the computational demands on neural networks can be daunting, even for minimal systems. For AI to be seamlessly integrated into everyday devices and essential systems, it must perform optimally within the constraints of limited resources while maintaining its efficacy.

Throughout the book, we have highlighted the importance of pursuing efficiency to ensure that AI models are streamlined, rapid, and sustainable. By optimizing models for efficiency, we can widen their applicability across various platforms and scenarios, enabling AI to be deployed in resource-constrained environments such as embedded systems and edge devices. This pursuit of efficiency is necessary for the widespread adoption and practical implementation of AI technologies in real-world applications.

20.6 ML Architecture Optimization

We then explored various model architectures, from the foundational perceptron to the sophisticated transformer networks, each tailored to specific tasks and data types. This exploration has showcased machine learning models' remarkable diversity and adaptability, enabling them to tackle various problems across domains.

However, when deploying these models on systems, especially resource-constrained embedded systems, model optimization becomes a necessity. The evolution of model architectures, from the early MobileNets designed for mobile devices to the more recent TinyML models optimized for microcontrollers, is a testament to the continued innovation.

In the chapter on model optimization, we looked into the art and science of optimizing machine learning models to ensure they are lightweight, efficient, and effective when deployed in TinyML scenarios. We explored techniques such as model compression, quantization, and architecture search, which allow us to reduce the computational footprint of models while maintaining their performance. By applying these optimization techniques, we can create models tailored to the specific constraints of embedded systems, enabling the deployment of powerful AI capabilities on edge devices. This opens many possibilities for intelligent, real-time processing and decision-making in IoT, robotics, and mobile computing applications. As we continue pushing the boundaries of AI efficiency, we expect to see even more innovative solutions for deploying machine learning models in resource-constrained environments.

20.7 AI Hardware Advancements

Over the years, we have witnessed remarkable strides in ML hardware, driven by the insatiable demand for computational power and the need to address the challenges of resource constraints in real-world deployments. These advancements have been crucial in enabling the deployment of powerful AI capabilities on devices with limited resources, opening up new possibilities across various industries.

Specialized hardware acceleration is essential to overcome these constraints and enable high-performance machine learning. Hardware accelerators, such as GPUs, FPGAs, and ASICs, optimize compute-intensive operations, particularly inference, by leveraging custom silicon designed for efficient matrix multiplications. These accelerators provide substantial speedups compared to general-purpose CPUs, enabling real-time execution of advanced ML models on devices with strict size, weight, and power limitations.

We have also explored the various techniques and approaches for hardware acceleration in embedded machine-learning systems. We discussed the tradeoffs in selecting the appropriate hardware for specific use cases and the importance of software optimizations to harness these accelerators' capabilities fully. By understanding these concepts, ML practitioners can make informed decisions when designing and deploying ML systems.

Given the plethora of ML hardware solutions available, benchmarking has become essential to developing and deploying machine learning systems. Benchmarking allows developers to measure and compare the performance of different hardware platforms, model architectures, training procedures, and deployment strategies. By utilizing well-established benchmarks like MLPerf, practitioners gain valuable insights into the most effective approaches for a given problem, considering the unique constraints of the target deployment environment.

Advancements in ML hardware, combined with insights gained from benchmarking and optimization techniques, have paved the way for successfully deploying machine learning capabilities on various devices, from powerful edge servers to resource-constrained microcontrollers. As the field continues to evolve, we expect to see even more innovative hardware solutions and benchmarking approaches that will further push the boundaries of what is possible with embedded machine learning systems.

20.8 On-Device Learning

In addition to the advancements in ML hardware, we also explored on-device learning, where models can adapt and learn directly on the device. This approach has significant implications for data privacy and security, as sensitive information can be processed locally without the need for transmission to external servers.

On-device learning enhances privacy by keeping data within the confines of the device, reducing the risk of unauthorized access or data breaches. It also reduces reliance on cloud connectivity, enabling ML models to function effectively even in scenarios with limited or intermittent internet access. We have discussed techniques such as transfer learning and federated learning, which have expanded the capabilities of on-device learning. Transfer learning allows models to leverage knowledge gained from one task or domain to improve performance on another, enabling more efficient and effective learning on resource-constrained devices. On the other hand, Federated learning enables collaborative model updates across distributed devices without centralized data aggregation. This approach allows multiple devices to contribute to learning while keeping their data locally, enhancing privacy and security.

These advancements in on-device learning have paved the way for more secure, privacy-preserving, and decentralized machine learning applications. As we prioritize data privacy and security in developing ML systems, we expect to see more innovative solutions that enable powerful AI capabilities while protecting sensitive information and ensuring user privacy.

20.9 ML Operation Streamlining

Even if we got the above pieces right, challenges and considerations must be addressed to ensure ML models' successful integration and operation in production environments. In the MLOps chapter, we studied the practices and architectures necessary to develop, deploy, and manage ML models throughout their entire lifecycle. We looked at the phases of ML, from data collection and model training to evaluation, deployment, and ongoing monitoring.

We learned about the importance of automation, collaboration, and continuous improvement in MLOps. By automating key processes, teams can streamline their workflows, reduce manual errors, and accelerate the deployment of ML models. Collaboration among diverse teams, including data scientists, engineers, and domain experts, ensures ML systems' successful development and deployment.

The ultimate goal of this chapter was to provide readers with a comprehensive understanding of ML model management, equipping them with the knowledge and tools necessary to build and run ML applications that deliver sustained value successfully. By adopting best practices in MLOps, organizations can ensure their ML initiatives' long-term success and impact, driving innovation and delivering meaningful results.

20.10 Security and Privacy

No ML system is ever complete without thinking about security and privacy. They are of major importance when developing real-world ML systems. As machine learning finds increasing application in sensitive domains such as healthcare, finance, and personal data, safeguarding confidentiality and preventing the misuse of data and models becomes a critical imperative, and these were the concepts we discussed previously. We examined security issues from multiple perspectives, starting with threats to models themselves, such as model theft and data poisoning. We also discussed the importance of hardware security, exploring topics like hardware bugs, physical attacks, and the unique security challenges faced by embedded devices.

In addition to security, we addressed the critical issue of data privacy. Techniques such as differential privacy were highlighted as tools to protect sensitive information. We also discussed the growing role of legislation in enforcing privacy protections, ensuring that user data is handled responsibly and transparently.

20.11 Ethical Considerations

As we embrace ML advancements in all facets of our lives, it is essential to remain mindful of the ethical considerations that will shape the future of AI. Fairness, transparency, accountability, and privacy in AI systems will be paramount as they become more integrated into our lives and decision-making processes.

As AI systems become more pervasive and influential, it is important to ensure that they are designed and deployed in a manner that upholds ethical principles. This means actively mitigating biases, promoting fairness, and preventing discriminatory outcomes. Additionally, ethical AI design ensures transparency in how AI systems make decisions, enabling users to understand and trust their outputs.

Accountability is another critical ethical consideration. As AI systems take on more responsibilities and make decisions that impact individuals and society, there must be clear mechanisms for holding these systems and their creators accountable. This includes establishing frameworks for auditing and monitoring AI systems and defining liability and redress mechanisms in case of harm or unintended consequences.

Ethical frameworks, regulations, and standards will be essential to address these ethical challenges. These frameworks should guide the responsible development and deployment of AI technologies, ensuring that they align with societal values and promote the well-being of individuals and communities.

Moreover, ongoing discussions and collaborations among researchers, practitioners, policymakers, and society will be important in navigating the ethical landscape of AI. These conversations should be inclusive and diverse, bringing together different perspectives and expertise to develop comprehensive and equitable solutions. As we move forward, it is the collective responsibility of all stakeholders to prioritize ethical considerations in the development and deployment of AI systems.

20.12 Sustainability

The increasing computational demands of machine learning, particularly for training large models, have raised concerns about their environmental impact due to high energy consumption and carbon emissions. As the scale and complexity of models continue to grow, addressing the sustainability challenges associated with AI development becomes imperative. To mitigate the environmental footprint of AI, the development of energy-efficient algorithms is necessary. This involves optimizing models and training procedures to minimize computational requirements while maintaining performance. Techniques such as model compression, quantization, and efficient neural architecture search can help reduce the energy consumption of AI systems.

Using renewable energy sources to power AI infrastructure is another important step towards sustainability. By transitioning to clean energy sources such as solar, wind, and hydropower, the carbon emissions associated with AI development can be significantly reduced. This requires a concerted effort from the AI community and support from policymakers and industry leaders to invest in and adopt renewable energy solutions. In addition, exploring alternative computing paradigms, such as neuromorphic and photonic computing, holds promise for developing more energy-efficient AI systems. By developing hardware and algorithms that emulate the brain’s processing mechanisms, we can potentially create AI systems that are both powerful and sustainable.

The AI community must prioritize sustainability as a key consideration in research and development. This involves investing in green computing initiatives, such as developing energy-efficient hardware and optimizing data centers for reduced energy consumption. It also requires collaboration across disciplines, bringing together AI, energy, and sustainability experts to develop holistic solutions.

Moreover, it is important to acknowledge that access to AI and machine learning compute resources may not be equally distributed across organizations and regions. This disparity can lead to a widening gap between those who have the means to leverage advanced AI technologies and those who do not. Organizations like the Organisation for Economic Cooperation and Development (OECD) are actively exploring ways to address this issue and promote greater equity in AI access and adoption. By fostering international cooperation, sharing best practices, and supporting capacity-building initiatives, we can ensure that AI's benefits are more widely accessible and that no one is left behind in the AI revolution.

20.13 Robustness and Resiliency

The chapter on Robust AI dives into the fundamental concepts, techniques, and tools for building fault-tolerant and error-resilient ML systems. In this chapter, we explored how, when developing machine learning systems, making them robust means accounting for hardware faults through techniques like redundant hardware, ensuring your model is resilient to issues like data poisoning and distribution shifts, and addressing software faults such as bugs, design flaws, and implementation errors.

By employing robust AI techniques, ML systems can maintain their reliability, safety, and performance even in adverse conditions. These techniques enable systems to detect and recover from faults, adapt to changing environments, and make decisions under uncertainty.

The chapter empowers researchers and practitioners to develop AI solutions that can withstand the complexities and uncertainties of real-world environments. It provides insights into the design principles, architectures, and algorithms underpinning robust AI systems and practical guidance on implementing and validating these systems.

20.14 Future of ML Systems

As we look to the future, the trajectory of ML systems points towards a paradigm shift from a model-centric approach to a more data-centric one. This shift recognizes that the quality and diversity of data are paramount to developing robust, reliable, and fair AI models.

We anticipate a growing emphasis on data curation, labeling, and augmentation techniques in the coming years. These practices aim to ensure that models are trained on high-quality, representative data that accurately reflects the complexities and nuances of real-world scenarios. By focusing on data quality and diversity, we can mitigate the risks of biased or skewed models that may perpetuate unfair or discriminatory outcomes.

This data-centric approach will be vital in addressing the challenges of bias, fairness, and generalizability in ML systems. By actively seeking out and incorporating diverse and inclusive datasets, we can develop more robust, equitable, and applicable models for various contexts and populations. Moreover, the emphasis on data will drive advancements in techniques such as data augmentation, where existing datasets are expanded and diversified through data synthesis, translation, and generation. These techniques can help overcome the limitations of small or imbalanced datasets, enabling the development of more accurate and generalizable models.

In recent years, generative AI has taken the field by storm, demonstrating remarkable capabilities in creating realistic images, videos, and text. However, the rise of generative AI also brings new challenges for ML systems. Unlike traditional ML systems, generative models often demand more computational resources and pose challenges in terms of scalability and efficiency. Furthermore, evaluating and benchmarking generative models presents difficulties, as traditional metrics used for classification tasks may not be directly applicable. Developing robust evaluation frameworks for generative models is an active area of research, and something we hope to write about soon!

Understanding and addressing these system challenges and ethical considerations will be important in shaping the future of generative AI and its impact on society. As ML practitioners and researchers, we are responsible for advancing the technical capabilities of generative models and developing robust systems and frameworks that can mitigate potential risks and ensure the beneficial application of this powerful technology.

20.15 AI for Good

The potential for AI to be used for social good is vast, provided that responsible ML systems are developed and deployed at scale across various use cases. To realize this potential, it is essential for researchers and practitioners to actively engage in the process of learning, experimentation, and pushing the boundaries of what is possible.

Throughout the development of ML systems, it is important to remember the key themes and lessons explored in this book. These include the importance of data quality and diversity, the pursuit of efficiency and robustness, the potential of TinyML and neuromorphic computing, and the imperative of security and privacy. These insights inform the work and guide the decisions of those involved in developing AI systems.

It is important to recognize that the development of AI is not solely a technical endeavor but also a deeply human one. It requires collaboration, empathy, and a commitment to understanding the societal implications of the systems being created. Engaging with experts from diverse fields, such as ethics, social sciences, and policy, is essential to ensure that the AI systems developed are technically sound, socially responsible, and beneficial. Embracing the opportunity to be part of this transformative field and shaping its future is a privilege and a responsibility. By working together, we can create a world where ML systems serve as tools for positive change and improving the human condition.

20.16 Congratulations

Congratulations on coming this far, and best of luck in your future endeavors! The future of AI is bright and filled with endless possibilities. It will be exciting to see the incredible contributions you will make to this field.

Feel free to reach out to me anytime at vj at eecs dot harvard dot edu.

– Prof. Vijay Janapa Reddi, Harvard University

LABS

Overview

Welcome to the hands-on labs section, where you'll explore deploying machine learning (ML) models onto real embedded devices, offering a practical introduction to ML systems. Unlike traditional approaches with large-scale models, these labs focus on interacting directly with both hardware and software. They help us showcase various sensor modalities across different application use cases. This approach provides valuable insights into the challenges and opportunities of deploying AI on real physical systems.

Learning Objectives

By completing these labs, we hope learners will:

Tip

- Gain proficiency in setting up and deploying ML models on supported devices, enabling you to tackle real-world ML deployment scenarios with confidence.
- Understand the steps involved in adapting and experimenting with ML models for different applications, allowing you to optimize performance and efficiency.
- Learn troubleshooting techniques specific to embedded ML deployments, equipping you with the skills to overcome common pitfalls and challenges.
- Acquire practical experience in deploying TinyML models on embedded devices, bridging the gap between theory and practice.
- Explore various sensor modalities and their applications, expanding your understanding of how ML can be leveraged in diverse domains.
- Foster an understanding of the real-world implications and challenges associated with ML system deployments, preparing you for future projects.

Target Audience

These labs are designed for:

- **Beginners** in the field of machine learning who have a keen interest in exploring the intersection of ML and embedded systems.
- **Developers and engineers** looking to apply ML models to real-world applications using low-power, resource-constrained devices.
- **Enthusiasts and researchers** who want to gain practical experience in deploying AI on edge devices and understand the unique challenges involved.

Supported Devices

We have included laboratory materials for three key devices that represent different hardware profiles and capabilities.

- **Nicla Vision**: Optimized for vision-based applications like image classification and object detection, ideal for compact, low-power use cases. It is also suitable for keyword spotting and motion detection tasks.
- **XIAO ESP32S3**: A versatile, compact board suitable for vision, keyword spotting, and motion detection tasks.
- **Grove Vision AI V2**: Equipped with a dedicated Neural Processing Unit (NPU), this device enables more advanced machine learning tasks with enhanced on-device inference capabilities, making it ideal for sophisticated computer vision and AI applications.
- **Raspberry Pi**: A flexible platform for more computationally intensive tasks, including small language models and various classification and detection applications.

Exercise	Nicla Vision	XIAO ESP32S3	Grove Vision AI V2	Raspberry Pi
Installation & Setup	✓	✓	✓	✓
Keyword Spotting (KWS)	✓	✓		
Image Classification	✓	✓	✓	✓
Object Detection	✓	✓	✓	✓
Motion Detection	✓	✓		
Small Language Models (SLM)				✓
Vision Language Models (VLM)				✓

Lab Structure

Each lab follows a structured approach:

1. **Introduction**: Explore the application and its significance in real-world scenarios.
2. **Setup**: Step-by-step instructions to configure the hardware and software environment.
3. **Deployment**: Guidance on training and deploying the pre-trained ML models on supported devices.
4. **Exercises**: Hands-on tasks to modify and experiment with model parameters.
5. **Discussion**: Analysis of results, potential improvements, and practical insights.

Recommended Lab Sequence

If you're new to embedded ML, we suggest starting with setup and keyword spotting before moving on to image classification and object detection. Raspberry Pi users can explore more advanced tasks, like small language models, after familiarizing themselves with the basics.

Troubleshooting and Support

If you encounter any issues during the labs, please consult the troubleshooting comments or refer to the FAQs provided within each lab. For further assistance, feel free to reach out to our support team or engage with the community forums.

Credits

Special credit and thanks to Prof. Marcelo Rovai for his valuable contributions to the development and continuous refinement of these labs.

Getting Started

Welcome to the exciting world of embedded machine learning and TinyML! In this hands-on lab series, you'll explore various projects demonstrating the power of running machine learning models on resource-constrained devices. Before diving into the projects, ensure you have the necessary hardware and software.

Hardware Requirements

To follow along with the hands-on labs, you'll need the following hardware:

1. **Arduino Nicla Vision board**
   - The Arduino Nicla Vision is a powerful, compact board designed for professional-grade computer vision and audio applications. It features a high-quality camera module, a digital microphone, and an IMU, making it suitable for demanding projects in industries such as robotics, automation, and surveillance.
   - Arduino Nicla Vision specifications
   - Arduino Nicla Vision pinout diagram
2. **XIAO ESP32S3 Sense board**
   - The Seeed Studio XIAO ESP32S3 Sense is a tiny, feature-packed board designed for makers, hobbyists, and students interested in exploring edge AI applications. It comes equipped with a camera, microphone, and IMU, making it easy to get started with projects such as image classification, keyword spotting, and motion detection.
   - XIAO ESP32S3 Sense specifications
   - XIAO ESP32S3 Sense pinout diagram
3. **Grove Vision AI V2 board**
   - The Seeed Studio Grove Vision AI V2 is a compact, low-power, yet powerful device. It is an MCU-based system powered by the Arm Cortex-M55 and vision AI module Ethos-U55. It supports TensorFlow and PyTorch frameworks and is compatible with the Arduino IDE. With the SenseCraft AI algorithm platform, trained machine learning (ML) models can be deployed to the sensor without the need for coding. It features a standard CSI interface, an onboard

   digital microphone, and an SD card slot, making it highly suitable for various embedded AI vision projects.
   - Grove Vision AI V2 features and hardware overview

4. **Raspberry Pi Single Board Computer**
   - The Raspberry Pi is a powerful and versatile single-board computer that has become an essential tool for engineers across various disciplines. Developed by the Raspberry Pi Foundation, these compact devices offer a unique combination of affordability, computational power, and extensive GPIO (General Purpose Input/Output) capabilities, making them ideal for prototyping, embedded systems development, and advanced engineering projects.
   - Raspberry Pi Hardware Documentation
   - Camera Documentation
5. **Additional accessories**
   - USB-C cable for programming and powering the XIAO and the Grove Vision AI V2
   - Micro-USB cable for programming and powering the Nicla
   - Power Supply for the Raspberries
   - Breadboard and jumper wires (optional, for connecting additional sensors)

The Arduino Nicla Vision is tailored for professional-grade applications, offering advanced features and performance suitable for demanding industrial projects. On the other hand, the Seeed Studio XIAO ESP32S3 Sense is geared toward makers, hobbyists, and students who want to explore edge AI applications in a more accessible and beginner-friendly format. Both boards have their strengths and target audiences, allowing users to choose the best fit for their needs and skill level. The Grove Vision AI V2 and the Raspberry Pi are aimed at more advanced engineering and machine learning projects.

Software Requirements

To program the boards and develop embedded machine learning projects, you'll need the following software:

1. **Arduino IDE**
   - Download and install
     - Install Arduino IDE
     - Follow the installation guide for your specific OS.
     - Arduino CLI
     - Configure the Arduino IDE for the Arduino Nicla Vision and XIAO ESP32S3 Sense boards.
2. **OpenMV IDE (optional)**
   - Download and install the OpenMV IDE for your operating system.

- Configure the OpenMV IDE for the Arduino Nicla Vision.

3. **Edge Impulse Studio**
    - Sign up for a free account on the Edge Impulse Studio.
    - Install Edge Impulse CLI
    - Follow the guides to connect your Arduino Nicla Vision and XIAO ESP32S3 Sense boards to Edge Impulse Studio.
4. **Raspberry Pi OS**

- Download and install the Raspberry Pi Imager

Network Connectivity

Some projects may require internet connectivity for data collection or model deployment. Ensure your development environment connection is stable through Wi-Fi or Ethernet. For the Raspberry Pi, having a Wi-Fi or Ethernet connection is necessary for remote operation without the necessity to plug in a monitor, keyboard, and mouse.

- For the Arduino Nicla Vision, you can use the onboard Wi-Fi module to connect to a wireless network.
- For the XIAO ESP32S3 Sense, you can use the onboard Wi-Fi module or connect an external Wi-Fi or Ethernet module using the available pins.
- For the Grove Vision AI V2, you can use the onboard Wi-Fi module on the Master Controller (for example the XIAO ESP32S3) or connect an external Wi-Fi or Ethernet module using the available pins.
- For the Raspberry Pi, you can use the onboard Wi-Fi module to connect an external Wi-Fi or Ethernet module using the available connector.

Conclusion

With your hardware and software set up, you're ready to embark on your embedded machine learning journey. The hands-on labs will guide you through various projects, covering topics like image classification, object detection, keyword spotting, and motion classification.

If you encounter any issues or have questions, don't hesitate to consult the troubleshooting guides or forums or seek support from the community.

Let's dive in and unlock the potential of ML on real (tiny) systems!

Nicla Vision

These labs provide a unique opportunity to gain practical experience with machine learning (ML) systems. Unlike working with large models requiring data center-scale resources, these exercises allow you to directly interact with hardware and software using TinyML. This hands-on approach gives you a tangible understanding of the challenges and opportunities in deploying AI, albeit at a tiny scale. However, the principles are largely the same as what you would encounter when working with larger systems.

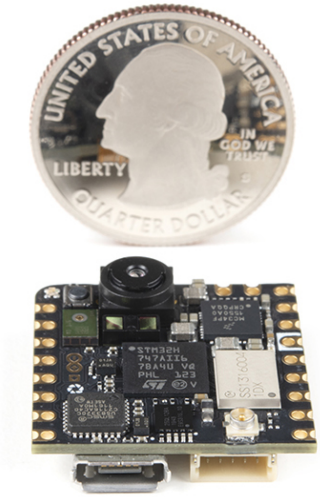


Figure 20.2: Nicla Vision. Source: Arduino

Pre-requisites

- **Nicla Vision Board**: Ensure you have the Nicla Vision board.
- **USB Cable**: For connecting the board to your computer.
- **Network**: With internet access for downloading necessary software.

Setup

- Setup Nicla Vision

Exercises

Modality	Task	Description	Link
Vision	Image Classification	Learn to classify images	Link
Vision	Object Detection	Implement object detection	Link
Sound	Keyword Spotting	Explore voice recognition systems	Link
IMU	Motion Classification and Anomaly Detection	Classify motion data and detect anomalies	Link

Setup

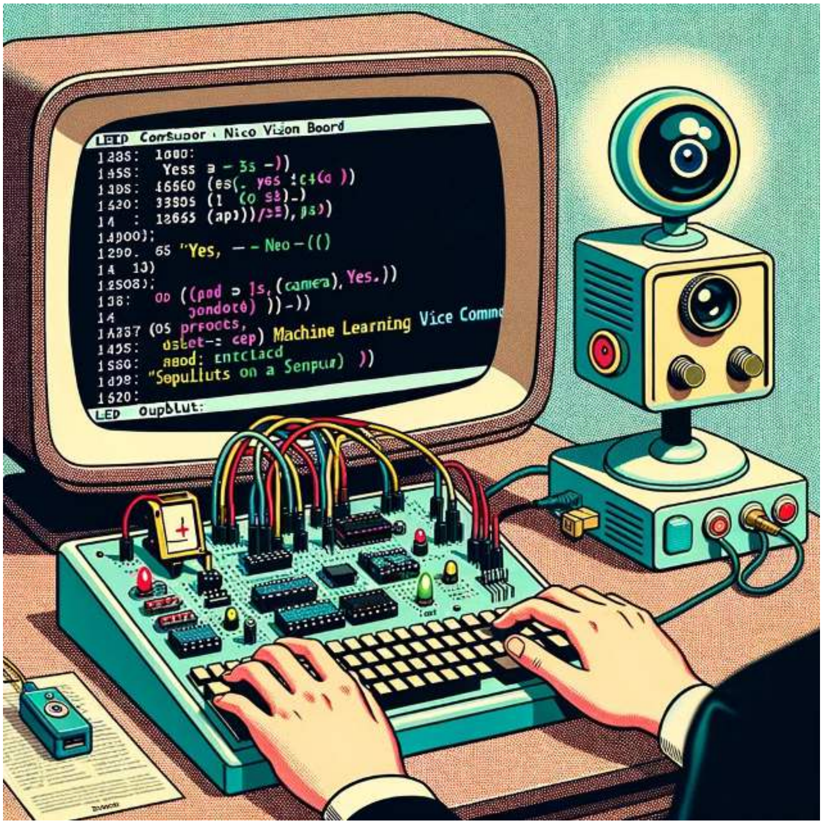
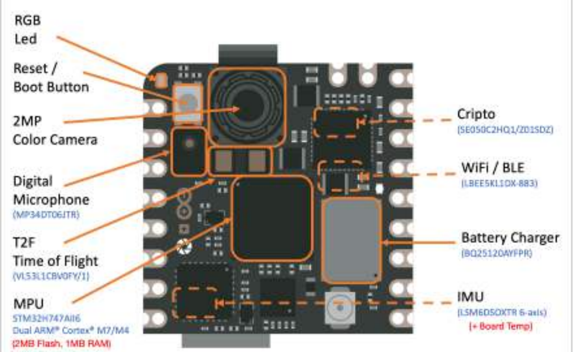


Figure 20.3: *DALL·E 3 Prompt: Illustration reminiscent of a 1950s cartoon where the Arduino NICLA VISION board, equipped with various sensors including a camera, is the focal point on an old-fashioned desk. In the background, a computer screen with rounded edges displays the Arduino IDE. The code is related to LED configurations and machine learning voice command detection. Outputs on the Serial Monitor explicitly display the words 'yes' and 'no'.*

Overview

The Arduino Nicla Vision (sometimes called *NiclaV*) is a development board that includes two processors that can run tasks in parallel. It is part of a family of development boards with the same form factor but designed for specific tasks, such as the Nicla Sense ME and the Nicla Voice. The *Niclas* can efficiently run processes created with TensorFlow Lite. For example, one of the cores of the NiclaV runs a computer vision algorithm on the fly (inference). At the same time, the other executes low-level operations like controlling a motor and communicating or acting as a user interface. The onboard wireless module allows the simultaneous management of WiFi and Bluetooth Low Energy (BLE) connectivity.

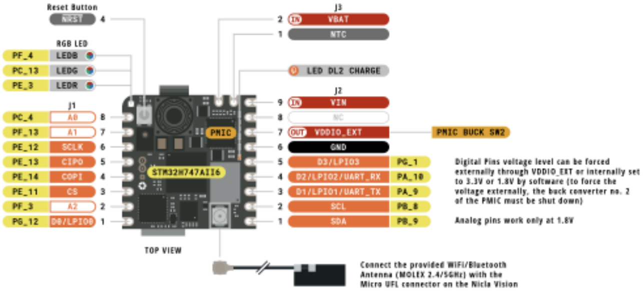


Hardware

Two Parallel Cores

The central processor is the dual-core STM32H747, including a Cortex M7 at 480 MHz and a Cortex M4 at 240 MHz. The two cores communicate via a Remote Procedure Call mechanism that seamlessly allows calling functions on the other processor. Both processors share all the on-chip peripherals and can run:

- Arduino sketches on top of the Arm Mbed OS
- Native Mbed applications
- MicroPython / JavaScript via an interpreter
- TensorFlow Lite



Digital Pins voltage level can be forced externally through VDDIO_EXT or internally set to 3.3V or 1.8V by software (to force the voltage externally, the buck converter no. 2 of the PMIC must be shut down)

Analog pins work only at 1.8V

Connect the provided WiFi/Bluetooth Antenna (MOLEX 2.4/5GHz) with the Micro UFL connector on the Nicla Vision

Memory

Memory is crucial for embedded machine learning projects. The NiclaV board can host up to 16 MB of QSPI Flash for storage. However, it is essential to consider that the MCU SRAM is the one to be used with machine learning inferences; the STM32H747 is only 1 MB, shared by both processors. This MCU also has incorporated 2 MB of FLASH, mainly for code storage.

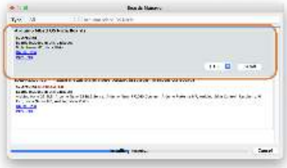
Sensors

- **Camera**: A GC2145 2 MP Color CMOS Camera.
- **Microphone**: The `MP34DT05` is an ultra-compact, low-power, omnidirectional, digital MEMS microphone built with a capacitive sensing element and the IC interface.
- **6-Axis IMU**: 3D gyroscope and 3D accelerometer data from the `LSM6DSOX` 6-axis IMU.
- **Time of Flight Sensor**: The `VL53L1CBV0FY` Time-of-Flight sensor adds accurate and low-power-ranging capabilities to Nicla Vision. The invisible near-infrared VCSEL laser (including the analog driver) is encapsulated with receiving optics in an all-in-one small module below the camera.

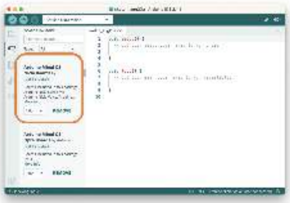
Arduino IDE Installation

Start connecting the board (*micro USB*) to your computer:

Install the Mbed OS core for Nicla boards in the Arduino IDE. Having the IDE open, navigate to `Tools > Board > Board Manager`, look for Arduino Nicla Vision on the search window, and install the board.



Arduino IDE 1.18



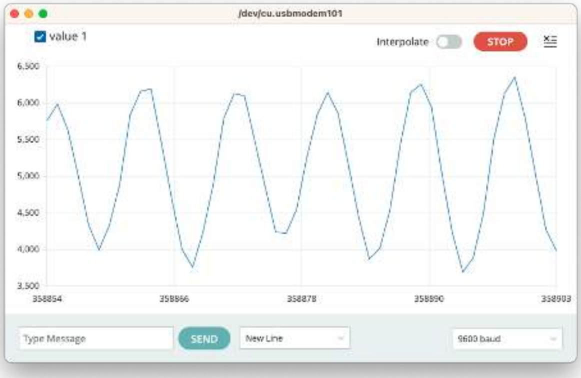
Arduino IDE 2.X

Next, go to `Tools > Board > Arduino Mbed OS Nicla Boards` and select `Arduino Nicla Vision`. Having your board connected to the USB, you should see the Nicla on Port and select it.

Open the Blink sketch on Examples/Basic and run it using the IDE Upload button. You should see the Built-in LED (green RGB) blinking, which means the Nicla board is correctly installed and functional!

Testing the Microphone

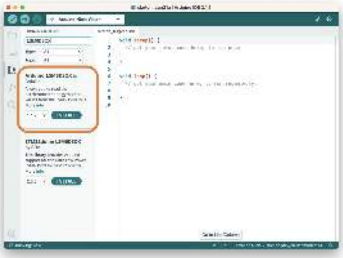
On Arduino IDE, go to `Examples > PDM > PDMSerialPlotter`, open it, and run the sketch. Open the Plotter and see the audio representation from the microphone:



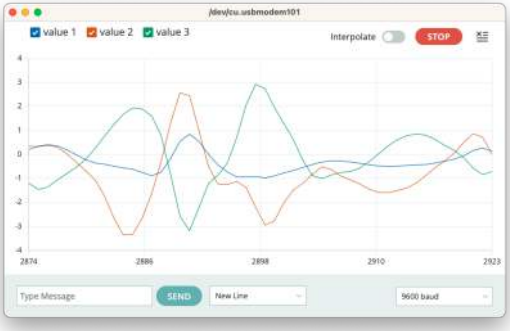
Vary the frequency of the sound you generate and confirm that the mic is working correctly.

Testing the IMU

Before testing the IMU, it will be necessary to install the LSM6DSOX library. To do so, go to Library Manager and look for LSM6DSOX. Install the library provided by Arduino:

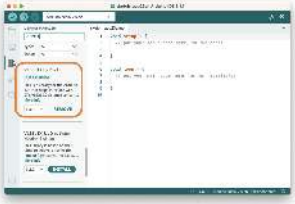


Next, go to `Examples > Arduino_LSM6DSOX > SimpleAccelerometer` and run the accelerometer test (you can also run Gyro and board temperature):

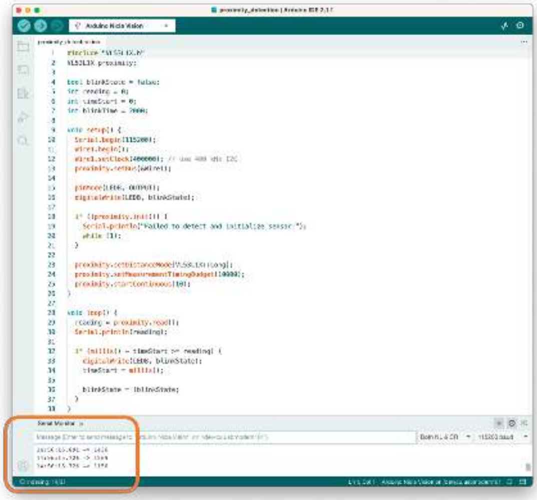


Testing the ToF (Time of Flight) Sensor

As we did with IMU, installing the VL53L1X ToF library is necessary. To do that, go to Library Manager and look for VL53L1X. Install the library provided by Pololu:



Next, run the sketch proximity_detection.ino:



```
#include "VL53L1X.h"
VL53L1X proximity;

bool blinkState = false;
int reading = 0;
int timeStart = 0;
int blinkTime = 2000;

void setup() {
  Serial.begin(115200);
  Wire1.begin();
  Wire1.setClock(400000); // use 400 kHz I2C
  proximity.setBus(&Wire1);

  pinMode(LEDB, OUTPUT);
  digitalWrite(LEDB, blinkState);

  if (!proximity.init()) {
    Serial.println("Failed to detect and initialize sensor!");
    while (1);
  }

  proximity.setDistanceMode(VL53L1X::Long);
  proximity.setMeasurementTimingBudget(10000);
  proximity.startContinuous(10);
}

void loop() {
  reading = proximity.read();
  Serial.println(reading);

  if (millis() - timeStart >= reading) {
    digitalWrite(LEDB, blinkState);
    timeStart = millis();

    blinkState = !blinkState;
  }
}
```

On the Serial Monitor, you will see the distance from the camera to an object in front of it (max of 4 m).

Testing the Camera

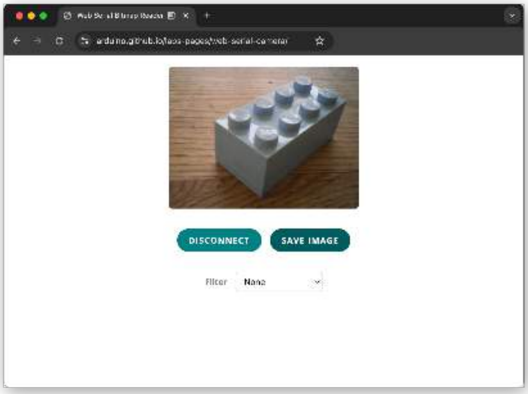
We can also test the camera using, for example, the code provided on `Examples > Camera > CameraCaptureRawBytes`. We cannot see the image directly, but we can get the raw image data generated by the camera.

We can use the `Web Serial Camera` (API) to see the image generated by the camera. This web application streams the camera image over Web Serial from camera-equipped Arduino boards.

The Web Serial Camera example shows you how to send image data over the wire from your Arduino board and how to unpack the data in JavaScript for rendering. In addition, in the source code of the web application, we can find some example image filters that show us how to manipulate pixel data to achieve visual effects.

The **Arduino sketch** (CameraCaptureWebSerial) for sending the camera image data can be found here and is also directly available from the "`Examples→Camera`" menu in the Arduino IDE when selecting the Nicla board.

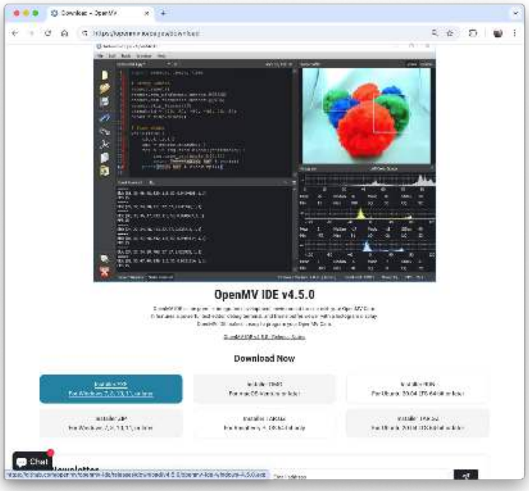
The **web application** for displaying the camera image can be accessed here. We may also look at [this tutorial, which explains the setup in more detail.



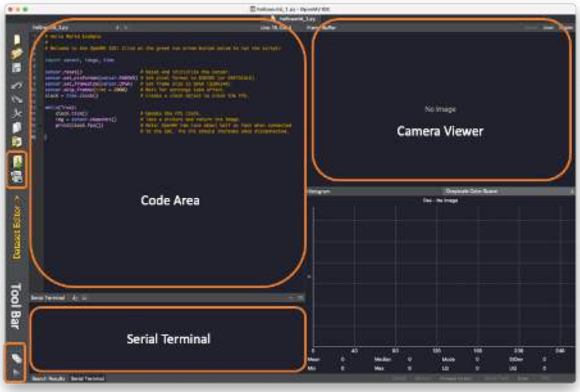
Installing the OpenMV IDE

OpenMV IDE is the premier integrated development environment with OpenMV cameras, similar to the Nicla Vision. It features a powerful text editor, debug terminal, and frame buffer viewer with a histogram display. We will use MicroPython to program the camera.

Go to the OpenMV IDE page, download the correct version for your Operating System, and follow the instructions for its installation on your computer.



The IDE should open, defaulting to the helloworld_1.py code on its Code Area. If not, you can open it from `Files > Examples > HelloWord > helloword.py`



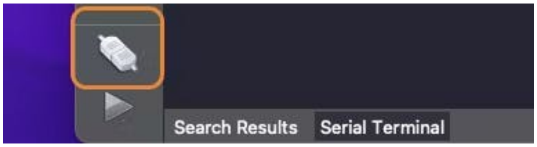
Any messages sent through a serial connection (using print() or error messages) will be displayed on the **Serial Terminal** during run time. The image captured by a camera will be displayed in the **Camera Viewer** Area (or Frame Buffer) and in the Histogram area, immediately below the Camera Viewer.

Updating the Bootloader

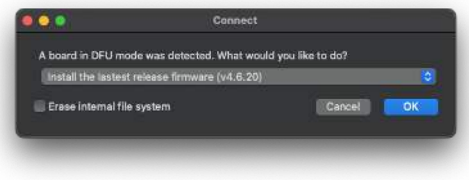
Before connecting the Nicla to the OpenMV IDE, ensure you have the latest bootloader version. Go to your Arduino IDE, select the Nicla board, and open the sketch on `Examples > STM_32H747_System STM32H747_manageBootloader`. Upload the code to your board. The Serial Monitor will guide you.

Installing the Firmware

After updating the bootloader, put the Nicla Vision in bootloader mode by double-pressing the reset button on the board. The built-in green LED will start fading in and out. Now return to the OpenMV IDE and click on the connect icon (Left ToolBar):



A pop-up will tell you that a board in DFU mode was detected and ask how you would like to proceed. First, select `Install the latest release firmware (vX.Y.Z)`. This action will install the latest OpenMV firmware on the Nicla Vision.



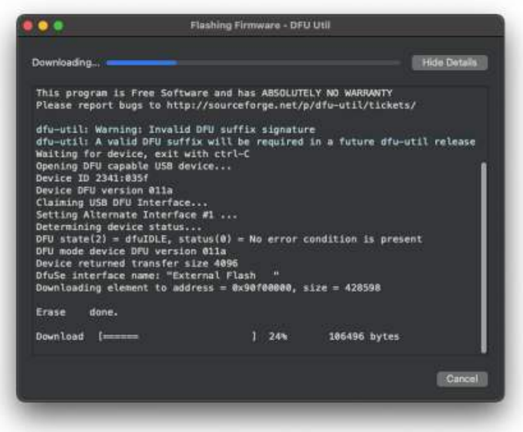
A board in DFU mode was detected. What would you like to do?

Install the lastest release firmware (v4.6.20)

□ Erase internal file system

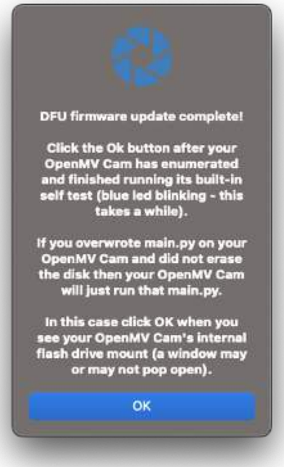
You can leave the option `Erase internal file system` unselected and click `[OK]`.

Nicla's green LED will start flashing while the OpenMV firmware is uploaded to the board, and a terminal window will then open, showing the flashing progress.



```
This program is Free Software and has ABSOLUTELY NO WARRANTY
Please report bugs to http://sourceforge.net/p/dfu-util/tickets/

dfu-util: Warning: Invalid DFU suffix signature
dfu-util: A valid DFU suffix will be required in a future dfu-util release
Waiting for device, exit with ctrl-C
Opening DFU capable USB device...
Device ID 2341:035f
Device DFU version 011a
Claiming USB DFU Interface...
Setting Alternate Interface #1 ...
Determining device status...
DFU state(2) = dfuIDLE, status(0) = No error condition is present
DFU mode device DFU version 011a
Device returned transfer size 4096
DfuSe interface name: "External Flash   "
Downloading element to address = 0x90f00000, size = 428598

Erase    done.

Download  [======                     ]  24%       106496 bytes
```

Wait until the green LED stops flashing and fading. When the process ends, you will see a message saying, "DFU firmware update complete!". Press `[OK]`.



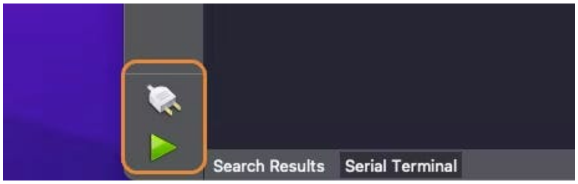
DFU firmware update complete!

Click the Ok button after your OpenMV Cam has enumerated and finished running its built-in self test (blue led blinking - this takes a while).

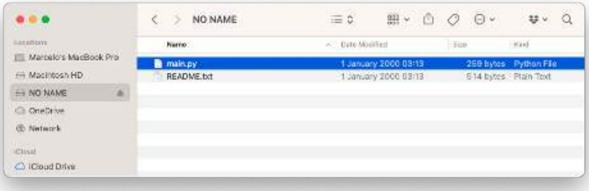
If you overwrote main.py on your OpenMV Cam and did not erase the disk then your OpenMV Cam will just run that main.py.

In this case click OK when you see your OpenMV Cam's internal flash drive mount (a window may or may not pop open).

A green play button appears when the Nicla Vison connects to the Tool Bar.

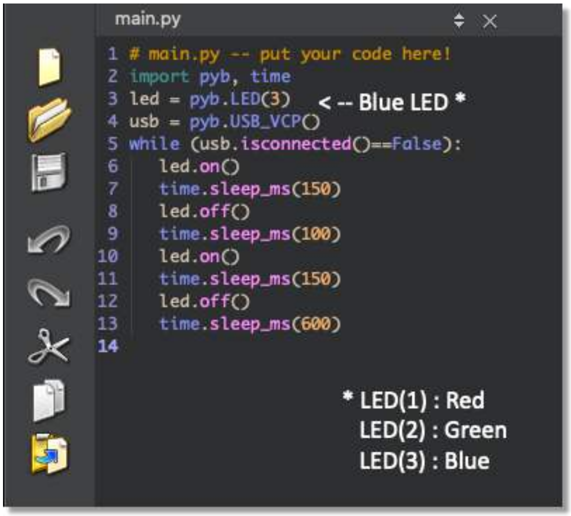


Also, note that a drive named “NO NAME” will appear on your computer.



Name	Date Modified	Size	Kind
main.py	1 January 2000 03:13	259 bytes	Python File
README.txt	1 January 2000 03:13	514 bytes	Plain Text

Every time you press the `[RESET]` button on the board, the main.py script stored on it automatically executes. You can load the main.py code on the IDE (`File > Open File...`).



```
# main.py -- put your code here!
import pyb, time
led = pyb.LED(3)
usb = pyb.USB_VCP()
while (usb.isconnected()==False):
    led.on()
    time.sleep_ms(150)
    led.off()
    time.sleep_ms(100)
    led.on()
    time.sleep_ms(150)
    led.off()
    time.sleep_ms(600)
```

<-- Blue LED *

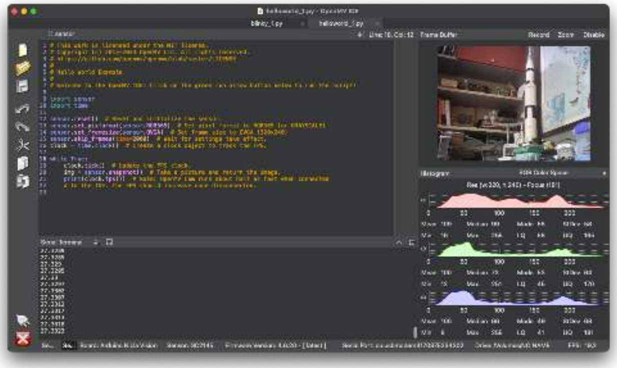
* LED(1) : Red
LED(2) : Green
LED(3) : Blue

This code is the "Blink" code, confirming that the HW is OK.

Testing the Camera

To test the camera, let's run *helloword_1.py*. For that, select the script on `File > Examples > HelloWorld > helloword.py`,

When clicking the green play button, the MicroPython script (*hellowolrd.py*) on the Code Area will be uploaded and run on the Nicla Vision. On-Camera Viewer, you will start to see the video streaming. The Serial Monitor will show us the FPS (Frames per second), which should be around 27fps.



Here is the `helloworld.py` script:

```
import sensor, time
```

```
sensor.reset()                      # Reset and initialize
                                    # the sensor.
sensor.set_pixformat(sensor.RGB565) # Set pixel format to RGB565
                                    # (or GRAYSCALE)
sensor.set_framesize(sensor.QVGA)   # Set frame size to
                                    # QVGA (320x240)
sensor.skip_frames(time = 2000)     # Wait for settings take
                                    # effect.
clock = time.clock()                # Create a clock object
                                    # to track the FPS.

while(True):
    clock.tick()                    # Update the FPS clock.
    img = sensor.snapshot()         # Take a picture and return
                                    # the image.
    print(clock.fps())
```

In GitHub, you can find the Python scripts used here.
The code can be split into two parts:

- **Setup**: Where the libraries are imported, initialized and the variables are defined and initiated.
- **Loop**: (while loop) part of the code that runs continually. The image (*img* variable) is captured (one frame). Each of those frames can be used for inference in Machine Learning Applications.

To interrupt the program execution, press the red `[X]` button.

> Note: OpenMV Cam runs about half as fast when connected to the IDE. The FPS should increase once disconnected.

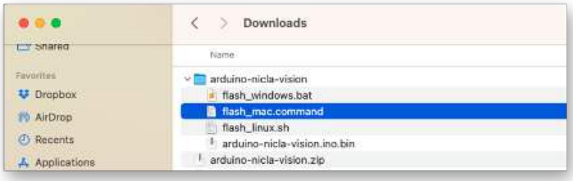
In the GitHub, You can find other Python scripts. Try to test the onboard sensors.

Connecting the Nicla Vision to Edge Impulse Studio

We will need the Edge Impulse Studio later in other labs. Edge Impulse is a leading development platform for machine learning on edge devices.

Edge Impulse officially supports the Nicla Vision. So, to start, please create a new project on the Studio and connect the Nicla to it. For that, follow the steps:

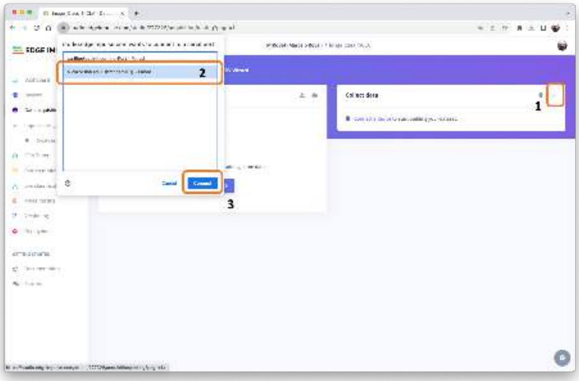
- Download the Arduino CLI for your specific computer architecture (OS)
- Download the most updated EI Firmware.
- Unzip both files and place all the files in the same folder.
- Put the Nicla-Vision on Boot Mode, pressing the reset button twice.
- Run the uploader (EI FW) corresponding to your OS:



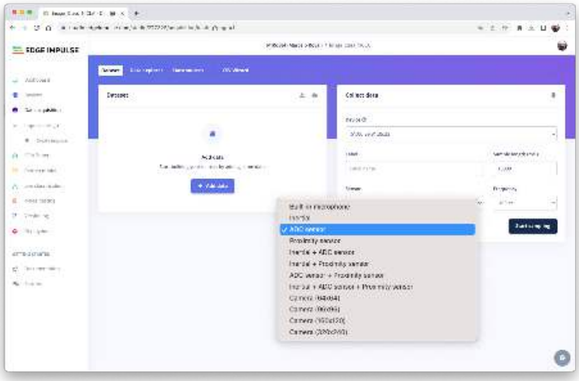
- Executing the specific batch code for your OS will upload the binary *arduino-nicla-vision.bin* to your board.

  Using `Chrome`, WebUSB can be used to connect the Nicla to the EI Studio. **The EI CLI is not needed.**

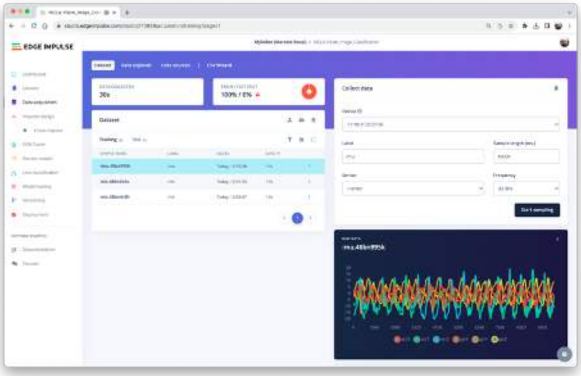
Go to your project on the Studio, and on the `Data Acquisition tab`, select `WebUSB` (1). A window will pop up; choose the option that shows that the `Nicla is paired` (2) and press `[Connect]` (3).



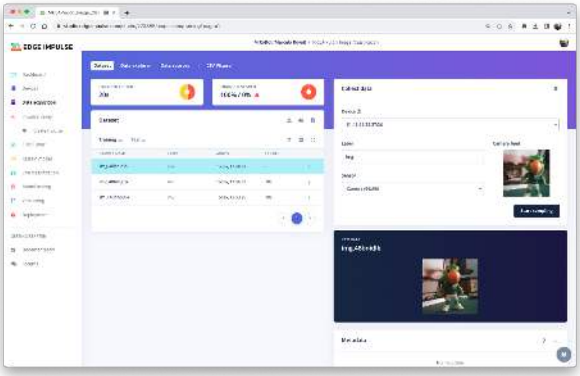
You can choose which sensor data to pick in the Collect Data section on the Data Acquisition tab.



For example. `IMU data (inercial)`:



Or Image (`Camera`):



You can also test an external sensor connected to the `ADC` (Nicla pin 0) and the other onboard sensors, such as the `built-in microphone`, the `ToF (Proximity)` or a combination of sensors (`fusion`).

Expanding the Nicla Vision Board (optional)

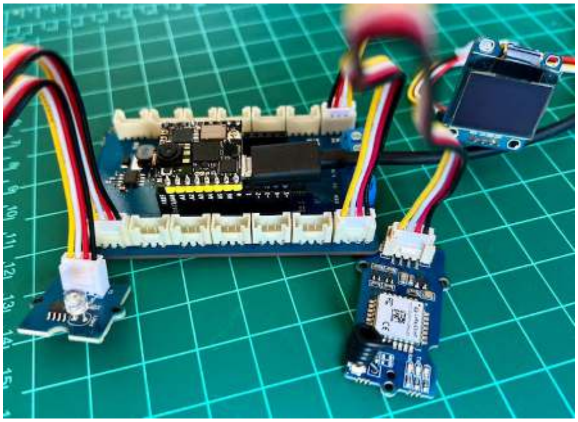
A last item to explore is that sometimes, during prototyping, it is essential to experiment with external sensors and devices. An excellent expansion to the Nicla is the Arduino MKR Connector Carrier (Grove compatible).

The shield has 14 Grove connectors: five single analog inputs (A0-A5), one double analog input (A5/A6), five single digital I/Os (D0-D4), one double digital I/O (D5/D6), one I2C (TWI), and one UART (Serial). All connectors are 5V compatible.

> Note that all 17 Nicla Vision pins will be connected to the Shield Groves, but some Grove connections remain disconnected.

This shield is MKR compatible and can be used with the Nicla Vision and Portenta.

For example, suppose that on a TinyML project, you want to send inference results using a LoRaWAN device and add information about local luminosity. Often, with offline operations, a local low-power display such as an OLED is advised. This setup can be seen here:

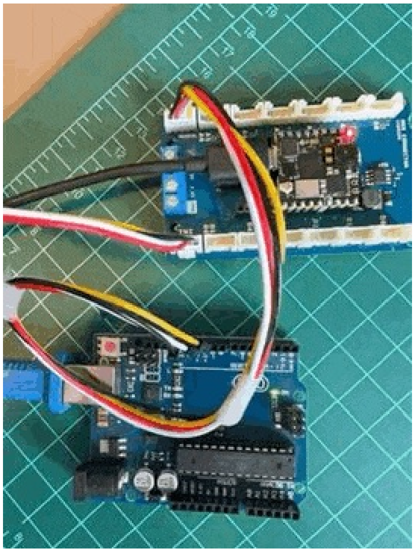


The Grove Light Sensor would be connected to one of the single Analog pins (A0/PC4), the LoRaWAN device to the UART, and the OLED to the I2C connector.

The Nicla Pins 3 (Tx) and 4 (Rx) are connected with the Serial Shield connector. The UART communication is used with the LoRaWan device. Here is a simple code to use the UART:

```
# UART Test - By: marcelo_rovai - Sat Sep 23 2023

import time
from pyb import UART
from pyb import LED

redLED = LED(1) # built-in red LED

# Init UART object.
# Nicla Vision's UART (TX/RX pins) is on "LP1"
uart = UART("LP1", 9600)

while(True):
    uart.write("Hello World!\r\n")
    redLED.toggle()
    time.sleep_ms(1000)
```

To verify that the UART is working, you should, for example, connect another device as the Arduino UNO, displaying "Hello Word" on the Serial Monitor. Here is the code.



Below is the *Hello World code* to be used with the I2C OLED. The MicroPython SSD1306 OLED driver (ssd1306.py), created by Adafruit, should also be uploaded to the Nicla (the ssd1306.py script can be found in GitHub).

```
# Nicla_OLED_Hello_World - By: marcelo_rovai - Sat Sep 30 2023

#Save on device: MicroPython SSD1306 OLED driver,
# I2C and SPI interfaces created by Adafruit
import ssd1306

from machine import I2C
i2c = I2C(1)

oled_width = 128
oled_height = 64
oled = ssd1306.SSD1306_I2C(oled_width, oled_height, i2c)

oled.text('Hello, World', 10, 10)
oled.show()
```

Finally, here is a simple script to read the ADC value on pin “PC4” (Nicla pin A0):

```
# Light Sensor (A0) - By: marcelo_rovai - Wed Oct 4 2023

import pyb
from time import sleep
```

```
adc = pyb.ADC(pyb.Pin("PC4"))   # create an analog object
                                # from a pin
val = adc.read()                # read an analog value

while (True):

    val = adc.read()
    print ("Light={}".format (val))
    sleep (1)
```

The ADC can be used for other sensor variables, such as Temperature.

> Note that the above scripts (downloaded from Github) introduce only how to connect external devices with the Nicla Vision board using MicroPython.

Conclusion

The Arduino Nicla Vision is an excellent *tiny device* for industrial and professional uses! However, it is powerful, trustworthy, low power, and has suitable sensors for the most common embedded machine learning applications such as vision, movement, sensor fusion, and sound.

> On the GitHub repository, you will find the last version of all the code used or commented on in this hands-on lab.

Resources

- Micropython codes
- Arduino Codes

Image Classification

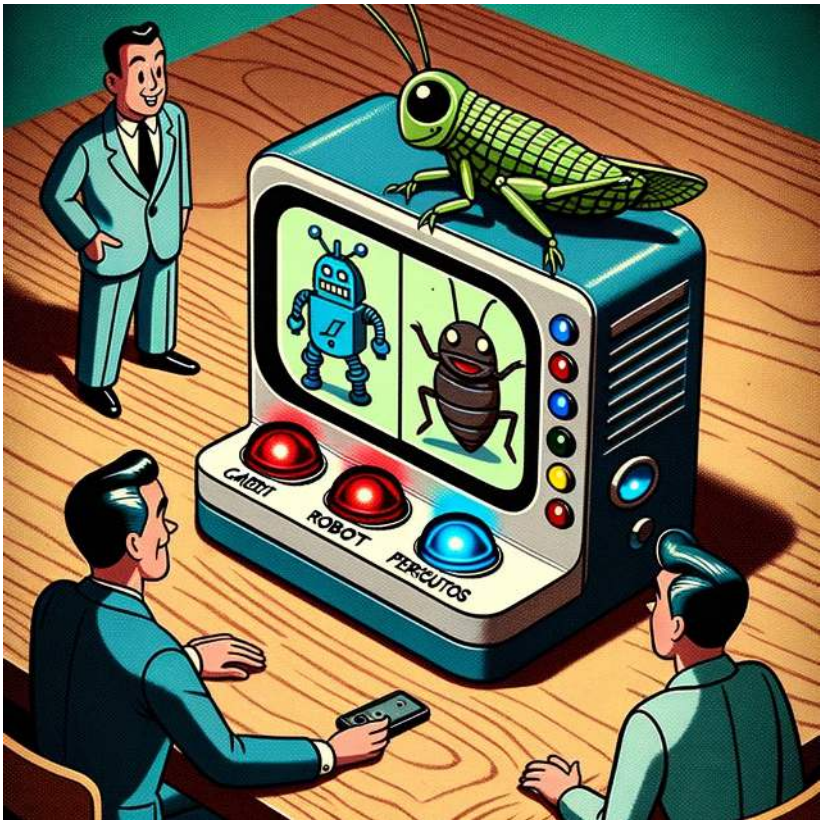
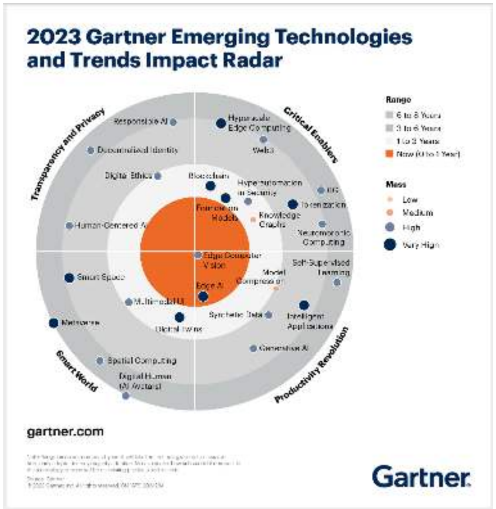


Figure 20.4: *DALL·E 3 Prompt: Cartoon in a 1950s style featuring a compact electronic device with a camera module placed on a wooden table. The screen displays blue robots on one side and green periquitos on the other. LED lights on the device indicate classifications, while characters in retro clothing observe with interest.*

Overview

As we initiate our studies into embedded machine learning or TinyML, it's impossible to overlook the transformative impact of Computer Vision (CV) and Artificial Intelligence (AI) in our lives. These two intertwined disciplines redefine what machines can perceive and accomplish, from autonomous vehicles and robotics to healthcare and surveillance.

More and more, we are facing an artificial intelligence (AI) revolution where, as stated by Gartner, **Edge AI** has a very high impact potential, and **it is for now**!



In the "bullseye" of the Radar is the *Edge Computer Vision*, and when we talk about Machine Learning (ML) applied to vision, the first thing that comes to mind is **Image Classification**, a kind of ML "Hello World"!

This lab will explore a computer vision project utilizing Convolutional Neural Networks (CNNs) for real-time image classification. Leveraging TensorFlow's robust ecosystem, we'll implement a pre-trained MobileNet model and adapt it for edge deployment. The focus will be optimizing the model to run efficiently on resource-constrained hardware without sacrificing accuracy.

We'll employ techniques like quantization and pruning to reduce the computational load. By the end of this tutorial, you'll have a working prototype capable of classifying images in real-time, all running on a low-power embedded system based on the Arduino Nicla Vision board.

Computer Vision

At its core, computer vision enables machines to interpret and make decisions based on visual data from the world, essentially mimicking the capability of the human optical system. Conversely, AI is a broader field encompassing machine learning, natural language processing, and robotics, among other technologies. When you bring AI algorithms into computer vision projects, you supercharge the system's ability to understand, interpret, and react to visual stimuli.

When discussing Computer Vision projects applied to embedded devices, the most common applications that come to mind are *Image Classification* and *Object Detection*.

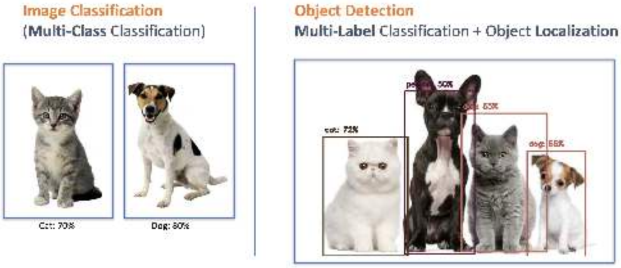
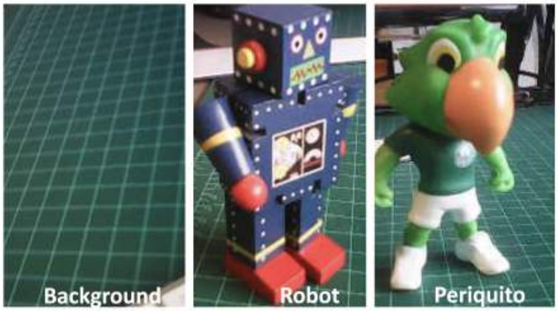


Image Classification
(Multi-Class Classification)
Cat: 70%
Dog: 80%
Object Detection
Multi-Label Classification + Object Localization
cat: 72%
50%
83%
dog: 88%

Both models can be implemented on tiny devices like the Arduino Nicla Vision and used on real projects. In this chapter, we will cover Image Classification.

Image Classification Project Goal

The first step in any ML project is to define the goal. In this case, the goal is to detect and classify two specific objects present in one image. For this project, we will use two small toys: a robot and a small Brazilian parrot (named Periquito). We will also collect images of a *background* where those two objects are absent.



Data Collection

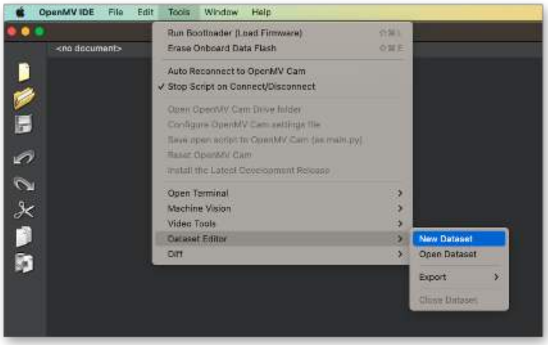
Once we have defined our Machine Learning project goal, the next and most crucial step is collecting the dataset. For image capturing, we can use:

- Web Serial Camera tool,
- Edge Impulse Studio,
- OpenMV IDE,
- A smartphone.

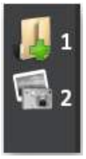
Here, we will use the **OpenMV IDE**.

Collecting Dataset with OpenMV IDE

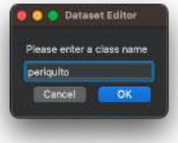
First, we should create a folder on our computer where the data will be saved, for example, "data." Next, on the OpenMV IDE, we go to `Tools > Dataset Editor` and select `New Dataset` to start the dataset collection:


OpenMV IDE
File
Edit
Tools
Window
Help
<no document>
Run Bootloader (Load Firmware)
Erase Onboard Data Flash
Auto Reconnect to OpenMV Cam
✓ Stop Script on Connect/Disconnect
Open OpenMV Cam Drive folder
Configure OpenMV Cam settings file
Save open script to OpenMV Cam (as main.py)
Reset OpenMV Cam
Install the Latest Development Release
Open Terminal
Machine Vision
Video Tools
Dataset Editor
Diff
New Dataset
Open Dataset
Export
Close Dataset

The IDE will ask us to open the file where the data will be saved. Choose the "data" folder that was created. Note that new icons will appear on the Left panel.



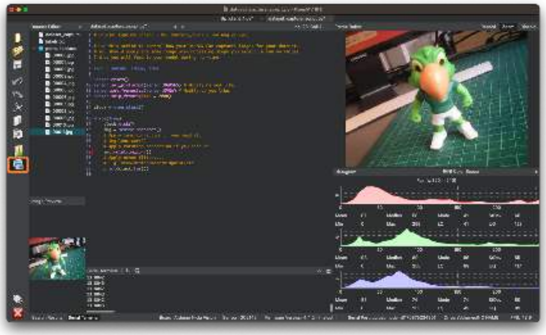
Using the upper icon (1), enter with the first class name, for example, "periquito":



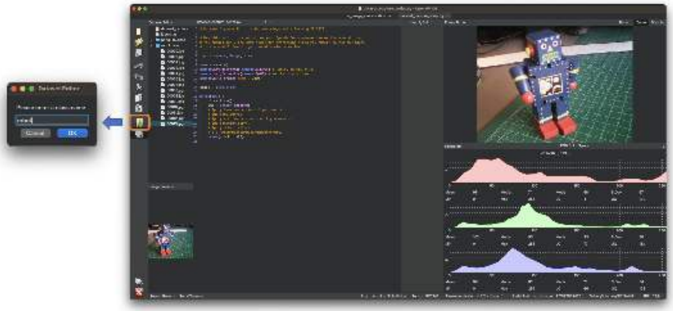
Please enter a class name

periquito

Running the `dataset_capture_script.py` and clicking on the camera icon (2) will start capturing images:



Repeat the same procedure with the other classes.

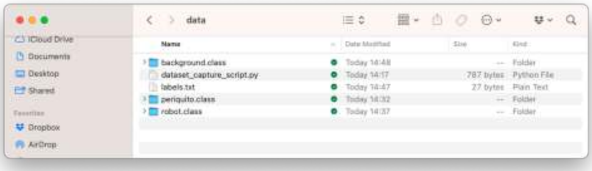


We suggest around 50 to 60 images from each category. Try to capture different angles, backgrounds, and light conditions.

The stored images use a QVGA frame size of 320 × 240 and the RGB565 (color pixel format).

After capturing the dataset, close the Dataset Editor Tool on the `Tools > Dataset Editor`.

We will end up with a dataset on our computer that contains three classes: *periquito*, *robot*, and *background*.

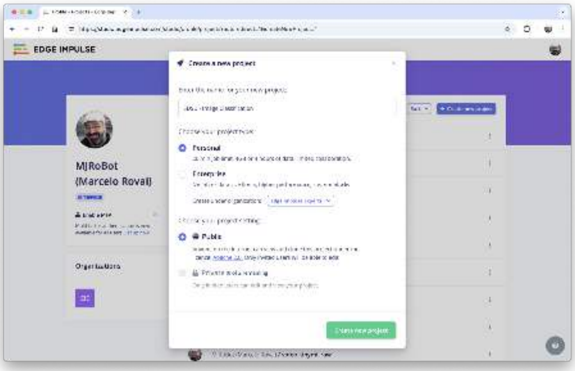


Name	Date Modified	Size	Kind
background.class	Today 14:48	--	Folder
dataset_capture_script.py	Today 14:17	787 bytes	Python File
labels.txt	Today 14:47	27 bytes	Plain Text
periquito.class	Today 14:32	--	Folder
robot.class	Today 14:37	--	Folder

We should return to *Edge Impulse Studio* and upload the dataset to our created project.

Training the model with Edge Impulse Studio

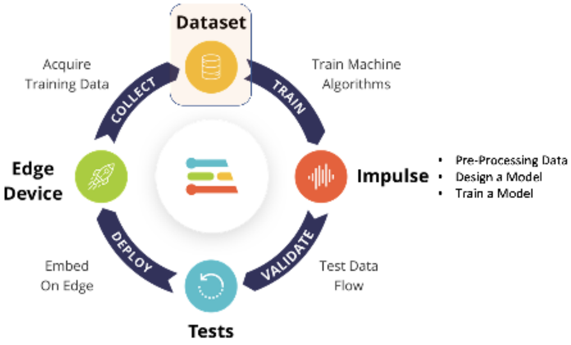
We will use the Edge Impulse Studio to train our model. Enter the account credentials and create a new project:



Here, you can clone a similar project: NICLA-Vision_Image_Classification.

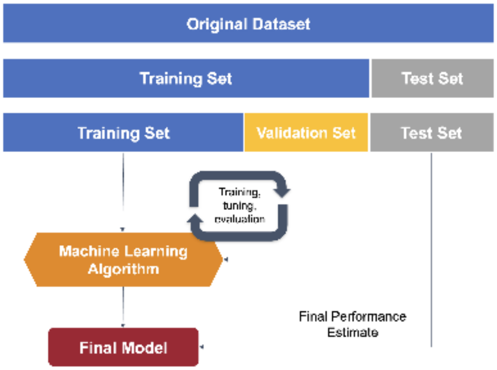
Dataset

Using the EI Studio (or *Studio*), we will go over four main steps to have our model ready for use on the Nicla Vision board: Dataset, Impulse, Tests, and Deploy (on the Edge Device, in this case, the NiclaV).

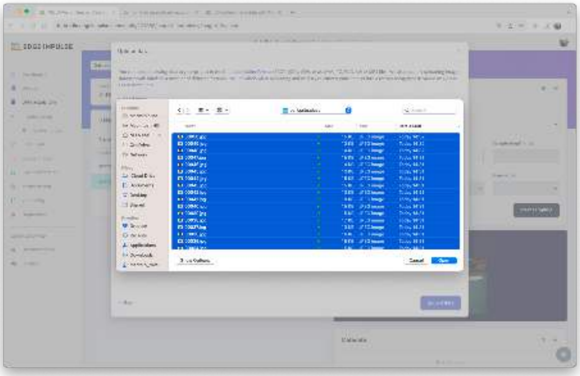


Regarding the Dataset, it is essential to point out that our Original Dataset, captured with the OpenMV IDE, will be split into *Training*, *Validation*, and *Test*. The Test Set will be spared from the beginning and reserved for use only in the Test phase after training. The Validation Set will be used during training.

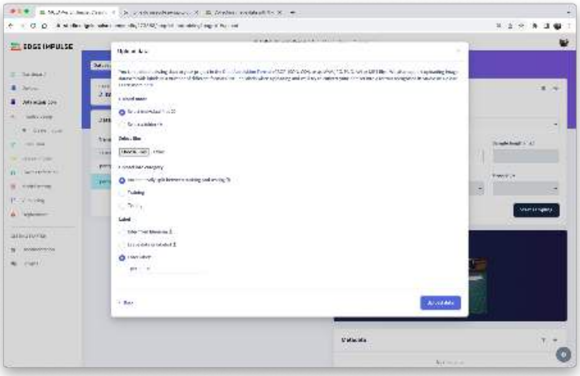
> The EI Studio will take a percentage of training data to be used for validation



On Studio, go to the `Data acquisition` tab, and on the UPLOAD DATA section, upload the chosen categories files from your computer:



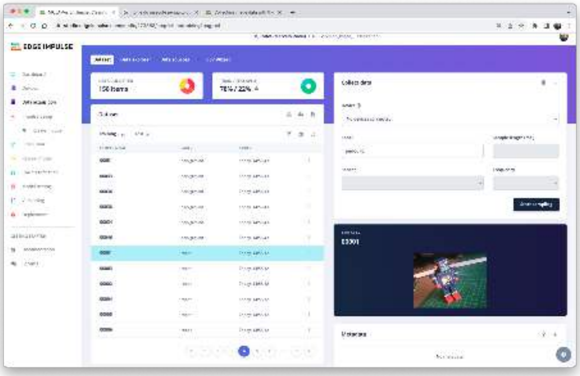
Leave to the Studio the splitting of the original dataset into *train and test* and choose the label about that specific data:



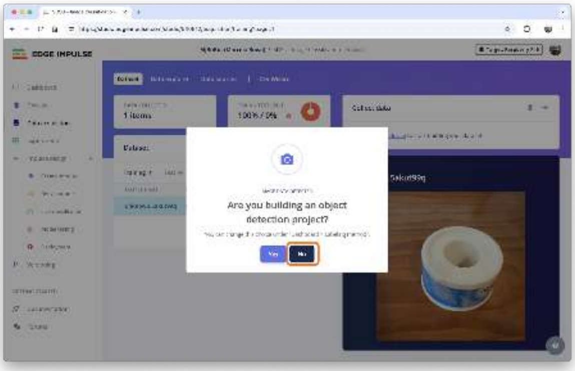
Repeat the procedure for all three classes.

Selecting a folder and upload all the files at once is possible.

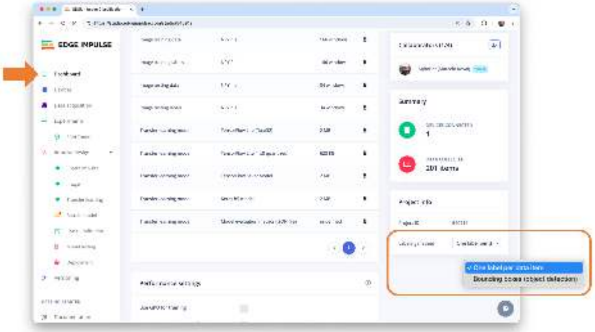
At the end, you should see your "raw data" in the Studio:



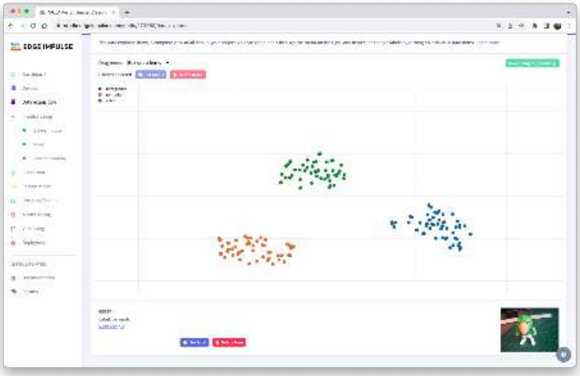
Note that when you start to upload the data, a pop-up window can appear, asking if you are building an Object Detection project. Select `[NO]`.



We can always change it in the Dashboard section: `One label per data item` (Image Classification):



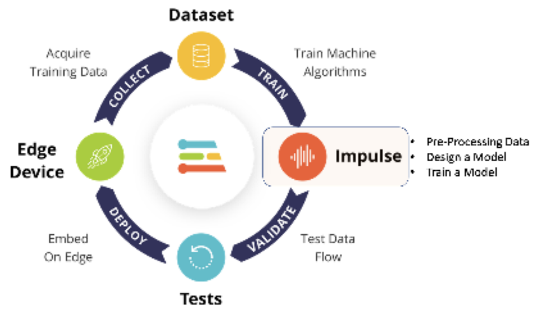
Optionally, the Studio allows us to explore the data, showing a complete view of all the data in the project. We can clear, inspect, or change labels by clicking on individual data items. In our case, the data seems OK.



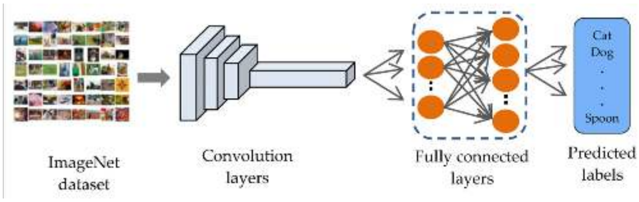
The Impulse Design

In this phase, we should define how to:

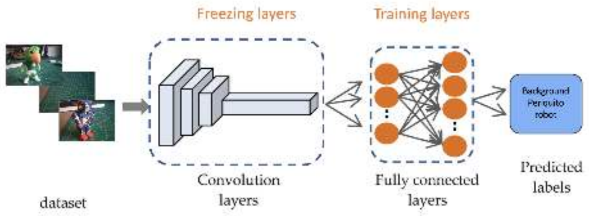
- Pre-process our data, which consists of resizing the individual images and determining the `color depth` to use (be it RGB or Grayscale) and
- Specify a Model, in this case, it will be the `Transfer Learning (Images)` to fine-tune a pre-trained MobileNet V2 image classification model on our data. This method performs well even with relatively small image datasets (around 150 images in our case).



Transfer Learning with MobileNet offers a streamlined approach to model training, which is especially beneficial for resource-constrained environments and projects with limited labeled data. MobileNet, known for its lightweight architecture, is a pre-trained model that has already learned valuable features from a large dataset (ImageNet).



By leveraging these learned features, you can train a new model for your specific task with fewer data and computational resources and yet achieve competitive accuracy.



This approach significantly reduces training time and computational cost, making it ideal for quick prototyping and deployment on embedded devices where efficiency is paramount.

Go to the Impulse Design Tab and create the *impulse*, defining an image size of 96x96 and squashing them (squared form, without cropping). Select Image and Transfer Learning blocks. Save the Impulse.

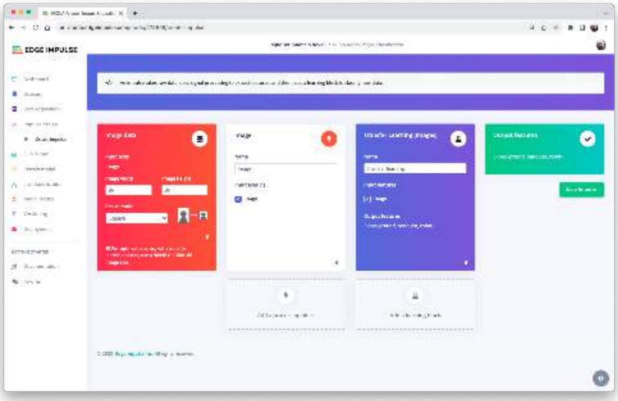
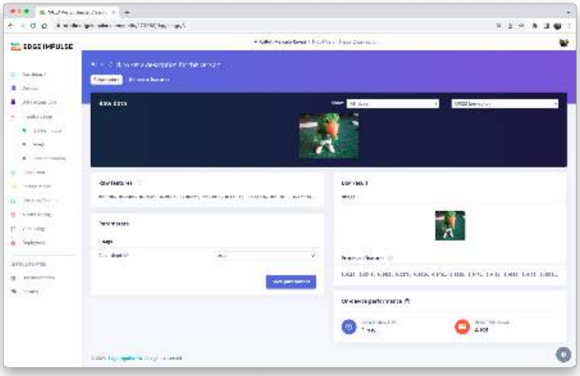
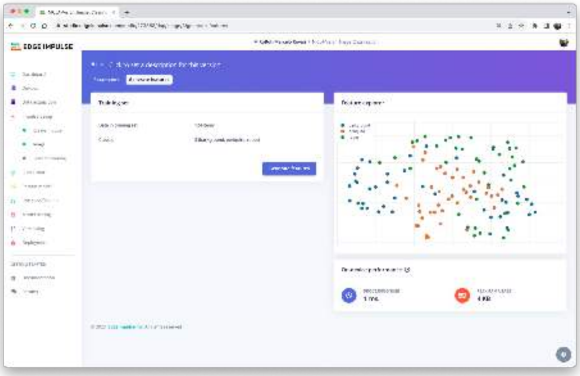


Image Pre-Processing

All the input QVGA/RGB565 images will be converted to 27,640 features ($96 \times 96 \times 3$).



Press [Save parameters] and Generate all features:



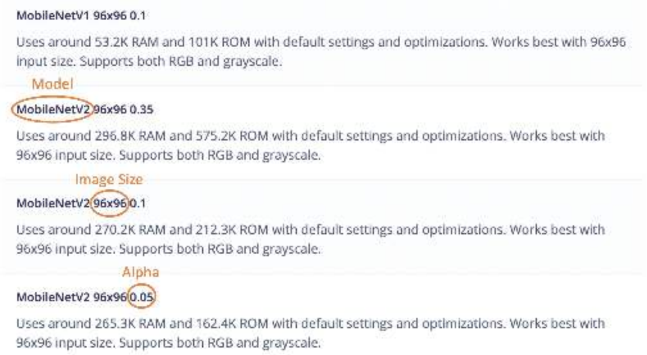
Model Design

In 2007, Google introduced MobileNetV1, a family of general-purpose computer vision neural networks designed with mobile devices in mind to support classification, detection, and more. MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of various use cases. in 2018, Google launched MobileNetV2: Inverted Residuals and Linear Bottlenecks.

MobileNet V1 and MobileNet V2 aim at mobile efficiency and embedded vision applications but differ in architectural complexity and performance. While both use depthwise separable convolutions to reduce the computational cost, MobileNet V2 introduces Inverted Residual Blocks and Linear Bottlenecks to improve performance. These new features allow V2 to capture more complex features using fewer parameters, making it computationally more efficient and generally more accurate than its predecessor. Additionally, V2 employs a non-linear activation in the intermediate expansion layer. It still uses a linear activation for the bottleneck layer, a design choice found to preserve important information through the network. MobileNet V2 offers an optimized architecture for higher accuracy and efficiency and will be used in this project.

Although the base MobileNet architecture is already tiny and has low latency, many times, a specific use case or application may require the model to be even smaller and faster. MobileNets introduces a straightforward parameter α (alpha) called width multiplier to construct these smaller, less computationally expensive models. The role of the width multiplier α is that of thinning a network uniformly at each layer.

Edge Impulse Studio can use both MobileNetV1 (96×96 images) and V2 (96×96 or 160×160 images), with several different α values (from 0.05 to 1.0). For example, you will get the highest accuracy with V2, 160×160 images, and $\alpha = 1.0$. Of course, there is a trade-off. The higher the accuracy, the more memory (around 1.3 MB RAM and 2.6 MB ROM) will be needed to run the model, implying more latency. The smaller footprint will be obtained at the other extreme with MobileNetV1 and $\alpha = 0.10$ (around 53.2 K RAM and 101 K ROM).



MobileNetV1 96x96 0.1

Uses around 53.2K RAM and 101K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

MobileNetV2 96x96 0.35

Uses around 296.8K RAM and 575.2K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

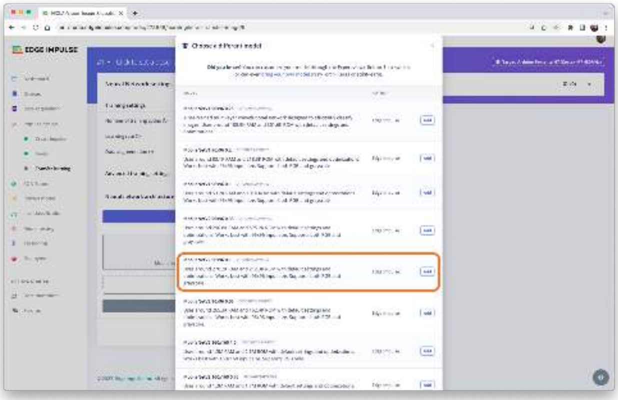
MobileNetV2 96x96 0.1

Uses around 270.2K RAM and 212.3K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

MobileNetV2 96x96 0.05

Uses around 265.3K RAM and 162.4K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

We will use **MobileNetV2 96x96 0.1** (**or 0.05**) for this project, with an estimated memory cost of 265.3 KB in RAM. This model should be OK for the Nicla Vision with 1MB of SRAM. On the Transfer Learning Tab, select this model:



Model Training

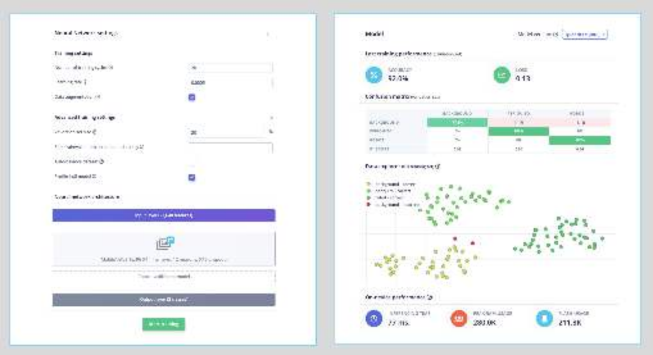
Another valuable technique to be used with Deep Learning is **Data Augmentation**. Data augmentation is a method to improve the accuracy of machine learning models by creating additional artificial data. A data augmentation system makes small, random changes to your training data during the training process (such as flipping, cropping, or rotating the images).

Looking under the hood, here you can see how Edge Impulse implements a data Augmentation policy on your data:

```
# Implements the data augmentation policy
def augment_image(image, label):
    # Flips the image randomly
    image = tf.image.random_flip_left_right(image)
    # Increase the image size, then randomly crop it down to
    # the original dimensions
    resize_factor = random.uniform(1, 1.2)
    new_height = math.floor(resize_factor * INPUT_SHAPE[0])
    new_width = math.floor(resize_factor * INPUT_SHAPE[1])
    image = tf.image.resize_with_crop_or_pad(image, new_height,
                                             new_width)
    image = tf.image.random_crop(image, size=INPUT_SHAPE)

    # Vary the brightness of the image
    image = tf.image.random_brightness(image, max_delta=0.2)

    return image, label
```

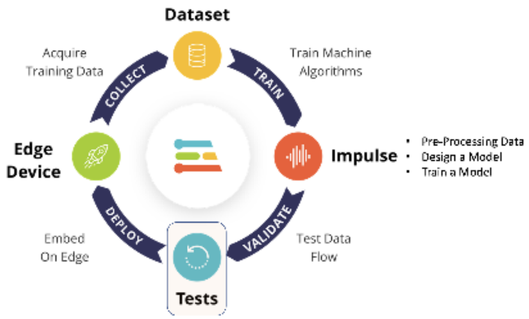
Exposure to these variations during training can help prevent your model from taking shortcuts by “memorizing” superficial clues in your training data, meaning it may better reflect the deep underlying patterns in your dataset.

The final layer of our model will have 12 neurons with a 15% dropout for overfitting prevention. Here is the Training result:

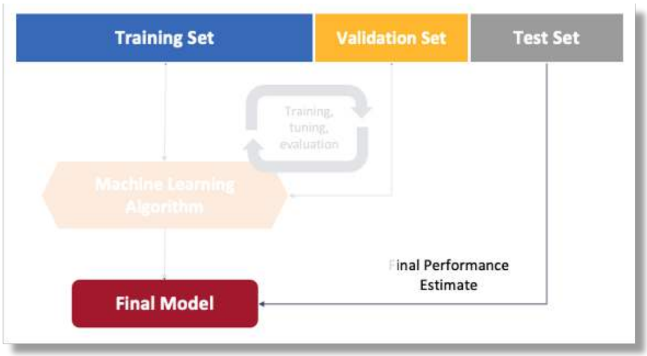


The result is excellent, with 77 ms of latency (estimated), which should result in around 13 fps (frames per second) during inference.

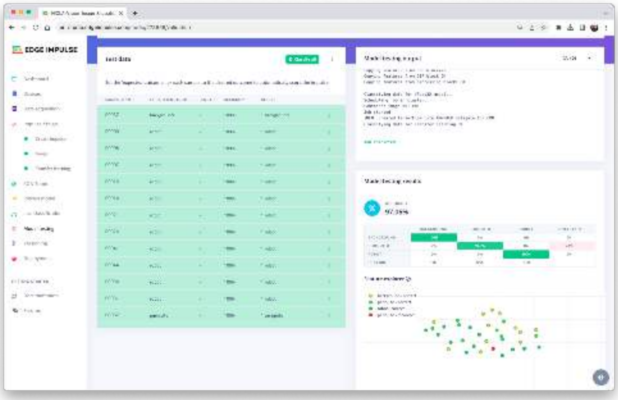
Model Testing



Now, we should take the data set put aside at the start of the project and run the trained model using it as input:

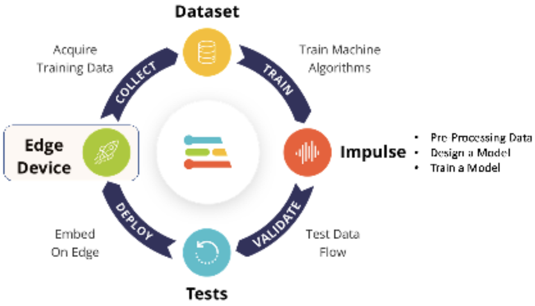


The result is, again, excellent.



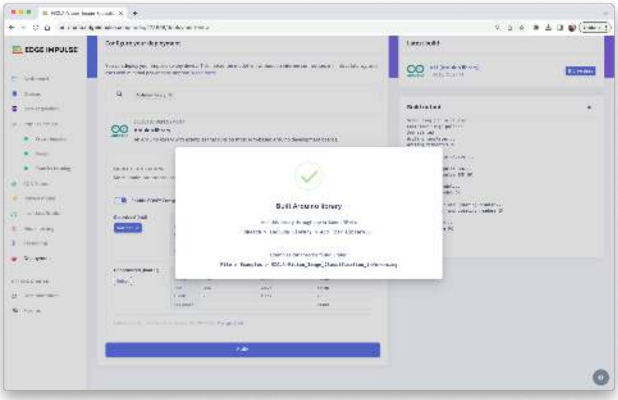
Deploying the model

At this point, we can deploy the trained model as a firmware (FW) and use the OpenMV IDE to run it using MicroPython, or we can deploy it as a C/C++ or an Arduino library.



Arduino Library

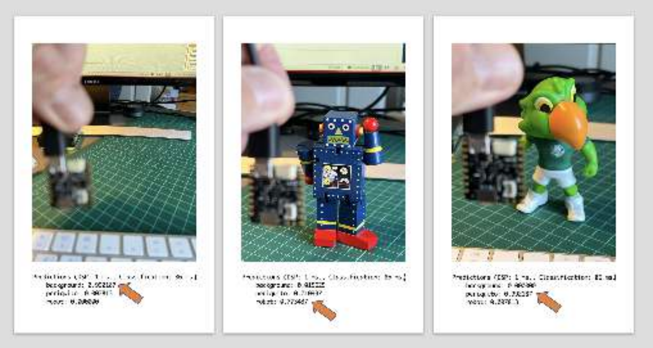
First, Let's deploy it as an Arduino Library:



We should install the library as.zip on the Arduino IDE and run the sketch *nicla_vision_camera.ino* available in Examples under the library name.

> Note that Arduino Nicla Vision has, by default, 512 KB of RAM allocated for the M7 core and an additional 244 KB on the M4 address space. In the code, this allocation was changed to 288 kB to guarantee that the model will run on the device (`malloc_addblock((void*)0x30000000, 288 * 1024);`).

The result is good, with 86 ms of measured latency.

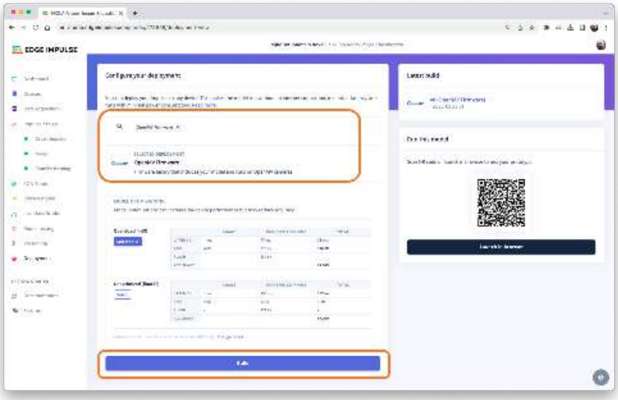


Here is a short video showing the inference results: https://youtu.be/bZPZZJblU-o

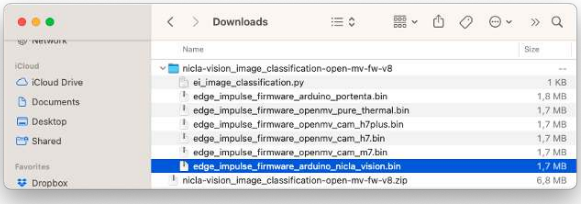
OpenMV

It is possible to deploy the trained model to be used with OpenMV in two ways: as a library and as a firmware (FW). Choosing FW, the Edge Impulse Studio generates optimized models, libraries, and frameworks needed to make the inference. Let's explore this option.

Select `OpenMV Firmware` on the `Deploy Tab` and press `[Build]`.

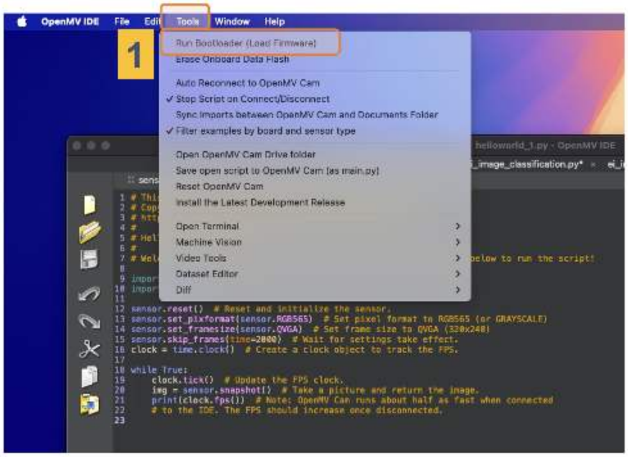


On the computer, we will find a ZIP file. Open it:



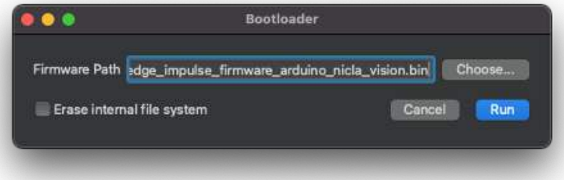
Name	Size
nicla-vision_image_classification-open-mv-fw-v8	--
ei_image_classification.py	1 KB
edge_impulse_firmware_arduino_portenta.bin	1,8 MB
edge_impulse_firmware_openmv_pure_thermal.bin	1,7 MB
edge_impulse_firmware_openmv_cam_h7plus.bin	1,7 MB
edge_impulse_firmware_openmv_cam_h7.bin	1,7 MB
edge_impulse_firmware_openmv_cam_m7.bin	1,7 MB
edge_impulse_firmware_arduino_nicla_vision.bin	1,7 MB
nicla-vision_image_classification-open-mv-fw-v8.zip	6,8 MB

Use the Bootloader tool on the OpenMV IDE to load the FW on your board (1):


OpenMV IDE
File
Edit
Tools
Window
Help
1
Run Bootloader (Load Firmware)
Erase Onboard Data Flash
Auto Reconnect to OpenMV Cam
✓ Stop Script on Connect/Disconnect
Sync Imports between OpenMV Cam and Documents Folder
✓ Filter examples by board and sensor type
Open OpenMV Cam Drive folder
Save open script to OpenMV Cam (as main.py)
Reset OpenMV Cam
Install the Latest Development Release
Open Terminal
Machine Vision
Video Tools
Dataset Editor
Diff
helloworld_1.py - OpenMV IDE
_image_classification.py*
ei_i

```
# Thi
# Cop
# htt
#
# Hel
#
# Wel                                                    below to run the script!

impor
impor

sensor.reset()  # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)  # Set pixel format to RGB565 (or GRAYSCALE)
sensor.set_framesize(sensor.QVGA)  # Set frame size to QVGA (320x240)
sensor.skip_frames(time=2000)  # Wait for settings take effect.
clock = time.clock()  # Create a clock object to track the FPS.

while True:
    clock.tick()  # Update the FPS clock.
    img = sensor.snapshot()  # Take a picture and return the image.
    print(clock.fps())  # Note: OpenMV Cam runs about half as fast when connected
    # to the IDE. The FPS should increase once disconnected.
```

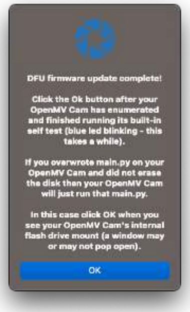
Select the appropriate file (.bin for Nicla-Vision):



Firmware Path: edge_impulse_firmware_arduino_nicla_vision.bin

□ Erase internal file system

After the download is finished, press OK:



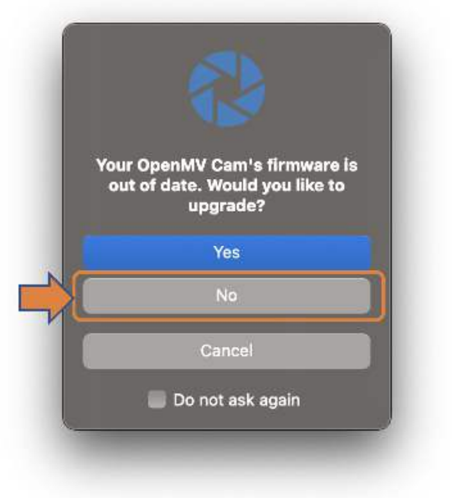
DFU firmware update complete!

Click the Ok button after your OpenMV Cam has enumerated and finished running its built-in self test (blue led blinking - this takes a while).

If you overwrote main.py on your OpenMV Cam and did not erase the disk then your OpenMV Cam will just run that main.py.

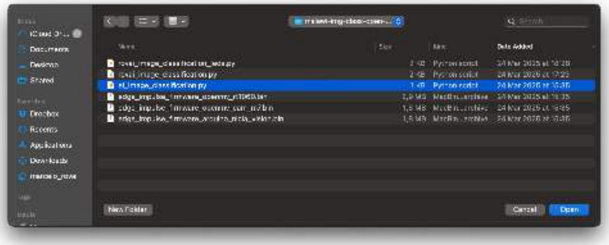
In this case click OK when you see your OpenMV Cam's internal flash drive mount (a window may or may not pop open).

If a message says that the FW is outdated, **DO NOT UPGRADE**. Select `[NO]`.



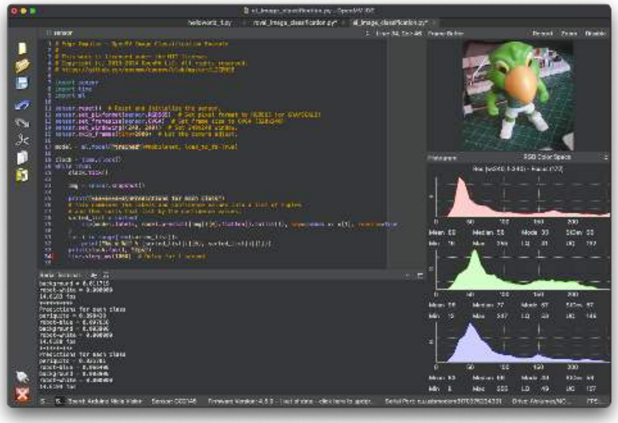
Your OpenMV Cam's firmware is out of date. Would you like to upgrade?

Now, open the script **ei_image_classification.py** that was downloaded from the Studio and the.bin file for the Nicla.



Name	Size	Kind	Date Added
rovai_image_classification_leds.py	2 KB	Python script	24 Mar 2025 at 18:28
rovai_image_classification.py	2 KB	Python script	24 Mar 2025 at 17:23
ei_image_classification.py	1 KB	Python script	24 Mar 2025 at 16:36
edge_impulse_firmware_openmv_rt1060.bin	2,9 MB	MacBin...archive	24 Mar 2025 at 16:35
edge_impulse_firmware_openmv_cam_m7.bin	1,8 MB	MacBin...archive	24 Mar 2025 at 16:35
edge_impulse_firmware_arduino_nicla_vision.bin	1,8 MB	MacBin...archive	24 Mar 2025 at 16:36

Run it. Pointing the camera to the objects we want to classify, the inference result will be displayed on the Serial Terminal.



The classification result will appear at the Serial Terminal. If it is difficult to read the result, include a new line in the code to add some delay:

```
import time
While True:
...
    time.sleep_ms(200)  # Delay for .2 second
```

Changing the Code to add labels

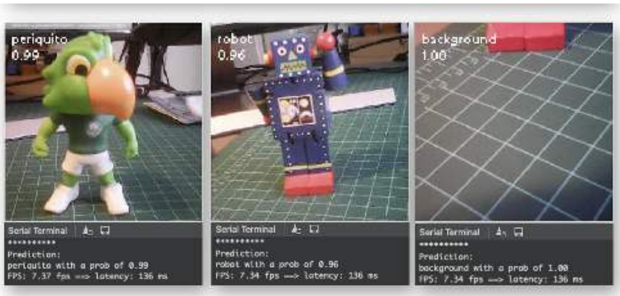
The code provided by Edge Impulse can be modified so that we can see, for test reasons, the inference result directly on the image displayed on the OpenMV IDE.

Upload the code from GitHub, or modify it as below:

```
# Marcelo Rovai - NICLA Vision - Image Classification
# Adapted from Edge Impulse - OpenMV Image Classification Example
# @24March25

import sensor
import time
import ml

sensor.reset()  # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)  # Set pixel format to RGB565 (or GRAYSCALE
sensor.set_framesize(sensor.QVGA)  # Set frame size to QVGA (320x240)
sensor.set_windowing((240, 240))  # Set 240x240 window.
sensor.skip_frames(time=2000)  # Let the camera adjust.

model = ml.Model("trained")#mobilenet, load_to_fb=True)
```

```
clock = time.clock()

while True:
    clock.tick()
    img = sensor.snapshot()

    fps = clock.fps()
    lat = clock.avg()
    print("**********\nPrediction:")
    # Combines labels & confidence into a list of tuples and then
    # sorts that list by the confidence values.
    sorted_list = sorted(
        zip(model.labels, model.predict([img])[0].flatten().tolist()),
      key=lambda x: x[1], reverse=True
    )

    # Print only the class with the highest probability
    max_val = sorted_list[0][1]
    max_lbl = sorted_list[0][0]

    if max_val < 0.5:
        max_lbl = 'uncertain'

    print("{} with a prob of {:.2f}".format(max_lbl, max_val))
    print("FPS: {:.2f} fps ==> latency: {:.0f} ms".format(fps, lat))

    # Draw the label with the highest probability to the image viewer
    img.draw_string(
    10, 10,
    max_lbl + "\n{:.2f}".format(max_val),
    mono_space = False,
    scale=3
    )

    time.sleep_ms(500)  # Delay for .5 second
```

Here you can see the result:



```
**********
Prediction:
periquito with a prob of 0.99
FPS: 7.37 fps ==> latency: 136 ms
```

```
**********
Prediction:
robot with a prob of 0.96
FPS: 7.34 fps ==> latency: 136 ms
```

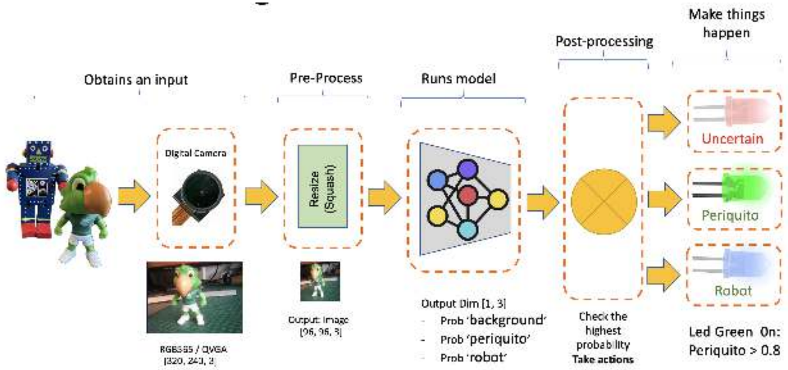
```
**********
Prediction:
background with a prob of 1.00
FPS: 7.34 fps ==> latency: 136 ms
```

Note that the latency (136 ms) is almost double of what we got directly with the Arduino IDE. This is because we are using the IDE as an interface and also the time to wait for the camera to be ready. If we start the clock just before the inference, the latency should drop to around 70 ms.

> The NiclaV runs about half as fast when connected to the IDE. The FPS should increase once disconnected.

Post-Processing with LEDs

When working with embedded machine learning, we are looking for devices that can continually proceed with the inference and result, taking some action directly on the physical world and not displaying the result on a connected computer. To simulate this, we will light up a different LED for each possible inference result.



To accomplish that, we should upload the code from GitHub or change the last code to include the LEDs:

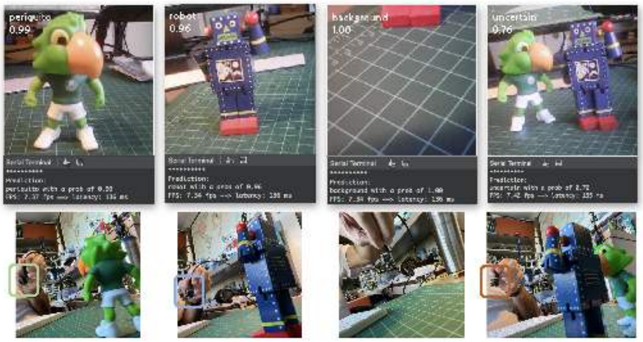
```
# Marcelo Rovai - NICLA Vision - Image Classification with LEDs
# Adapted from Edge Impulse - OpenMV Image Classification Example
# @24Aug23

import sensor, time, ml
from machine import LED
```

```
ledRed = LED("LED_RED")
ledGre = LED("LED_GREEN")
ledBlu = LED("LED_BLUE")

sensor.reset()  # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)  # Set pixel format to RGB565 (or GRAYSCALE)
sensor.set_framesize(sensor.QVGA)  # Set frame size to QVGA (320x240)
sensor.set_windowing((240, 240))  # Set 240x240 window.
sensor.skip_frames(time=2000)  # Let the camera adjust.

model = ml.Model("trained")#mobilenet, load_to_fb=True)

ledRed.off()
ledGre.off()
ledBlu.off()

clock = time.clock()

def setLEDs(max_lbl):
    if max_lbl == 'uncertain’:
        ledRed.on()
        ledGre.off()
        ledBlu.off()

    if max_lbl == 'periquito’:
        ledRed.off()
        ledGre.on()
        ledBlu.off()

    if max_lbl == 'robot’:
        ledRed.off()
        ledGre.off()
        ledBlu.on()

    if max_lbl == 'background’:
        ledRed.off()
        ledGre.off()
        ledBlu.off()

  while True:
    img = sensor.snapshot()

    clock.tick()
    fps = clock.fps()
    lat = clock.avg()
    print("**********\nPrediction:")
    sorted_list = sorted(
        zip(model.labels, model.predict([img])[0].flatten().tolist()),
      key=lambda x: x[1], reverse=True
    )

    # Print only the class with the highest probability
    max_val = sorted_list[0][1]
    max_lbl = sorted_list[0][0]

    if max_val < 0.5:
        max_lbl = 'uncertain'

    print("{} with a prob of {:.2f}".format(max_lbl, max_val))
    print("FPS: {:.2f} fps ==> latency: {:.0f} ms".format(fps, lat))

    # Draw the label with the highest probability to the image viewer
    img.draw_string(
    10, 10,
    max_lbl + "\n{:.2f}".format(max_val),
    mono_space = False,
    scale=3
    )

    setLEDs(max_lbl)
    time.sleep_ms(200)  # Delay for .2 second
```

Now, each time that a class scores a result greater than 0.8, the correspondent LED will be lit:

- Led Red 0n: Uncertain (no class is over 0.8)
- Led Green 0n: Periquito > 0.8
- Led Blue 0n: Robot > 0.8
- All LEDs Off: Background > 0.8

Here is the result:



```
**********
Prediction:
periquito with a prob of 0.99
FPS: 7.37 fps --> latency: 136 ms
```

```
**********
Prediction:
robot with a prob of 0.96
FPS: 7.34 fps --> latency: 136 ms
```

```
**********
Prediction:
background with a prob of 1.00
FPS: 7.34 fps --> latency: 136 ms
```

```
**********
Prediction:
uncertain with a prob of 0.76
FPS: 7.42 fps --> latency: 135 ms
```

In more detail

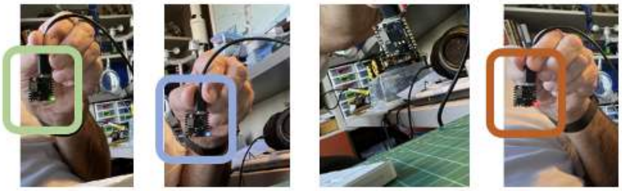
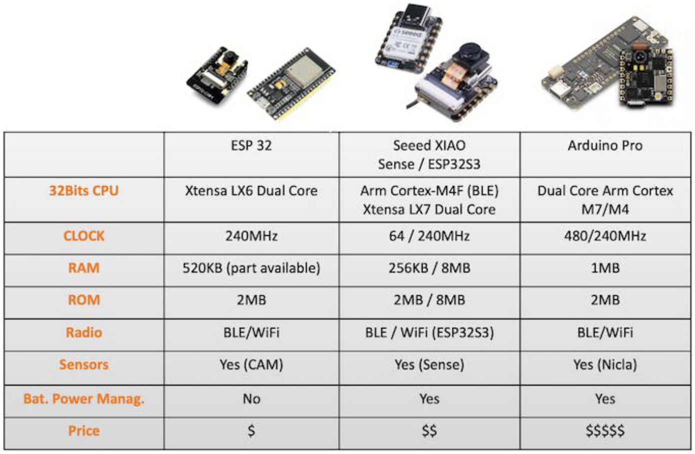


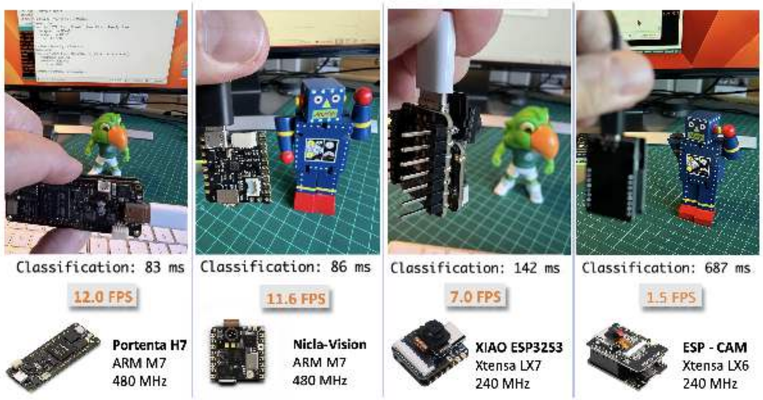
Image Classification (non-official) Benchmark

Several development boards can be used for embedded machine learning (TinyML), and the most common ones for Computer Vision applications (consuming low energy), are the ESP32 CAM, the Seeed XIAO ESP32S3 Sense, the Arduino Nicla Vison, and the Arduino Portenta.



	ESP 32	Seeed XIAO Sense / ESP32S3	Arduino Pro
32Bits CPU	Xtensa LX6 Dual Core	Arm Cortex-M4F (BLE) Xtensa LX7 Dual Core	Dual Core Arm Cortex M7/M4
CLOCK	240MHz	64 / 240MHz	480/240MHz
RAM	520KB (part available)	256KB / 8MB	1MB
ROM	2MB	2MB / 8MB	2MB
Radio	BLE/WiFi	BLE / WiFi (ESP32S3)	BLE/WiFi
Sensors	Yes (CAM)	Yes (Sense)	Yes (Nicla)
Bat. Power Manag.	No	Yes	Yes
Price	$	$$	$$$$$

Catching the opportunity, the same trained model was deployed on the ESP-CAM, the XIAO, and the Portenta (in this one, the model was trained again, using grayscaled images to be compatible with its camera). Here is the result, deploying the models as Arduino's Library:



Conclusion

Before we finish, consider that Computer Vision is more than just image classification. For example, you can develop Edge Machine Learning projects around vision in several areas, such as:

- **Autonomous Vehicles**: Use sensor fusion, lidar data, and computer vision algorithms to navigate and make decisions.
- **Healthcare**: Automated diagnosis of diseases through MRI, X-ray, and CT scan image analysis
- **Retail**: Automated checkout systems that identify products as they pass through a scanner.
- **Security and Surveillance**: Facial recognition, anomaly detection, and object tracking in real-time video feeds.
- **Augmented Reality**: Object detection and classification to overlay digital information in the real world.
- **Industrial Automation**: Visual inspection of products, predictive maintenance, and robot and drone guidance.
- **Agriculture**: Drone-based crop monitoring and automated harvesting.
- **Natural Language Processing**: Image captioning and visual question answering.
- **Gesture Recognition**: For gaming, sign language translation, and human-machine interaction.
- **Content Recommendation**: Image-based recommendation systems in e-commerce.

Resources

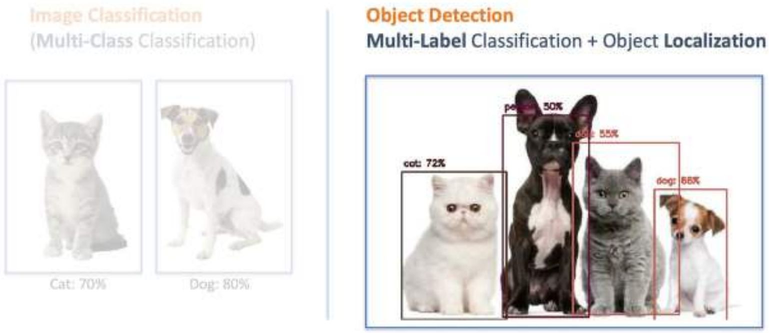
- Micropython codes
- Dataset
- Edge Impulse Project

Object Detection

Figure 20.5: *DALL·E 3 Prompt: Cartoon in the style of the 1940s or 1950s showcasing a spacious industrial warehouse interior. A conveyor belt is prominently featured, carrying a mixture of toy wheels and boxes. The wheels are distinguishable with their bright yellow centers and black tires. The boxes are white cubes painted with alternating black and white patterns. At the end of the moving conveyor stands a retro-styled robot, equipped with tools and sensors, diligently classifying and counting the arriving wheels and boxes. The overall aesthetic is reminiscent of mid-century animation with bold lines and a classic color palette.*

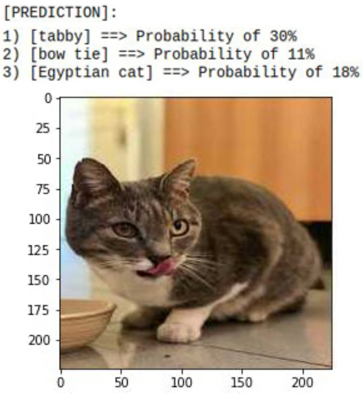
Overview

This continuation of Image Classification on Nicla Vision is now exploring **Object Detection**.



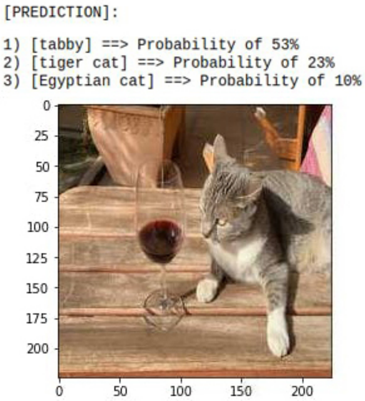
Object Detection versus Image Classification

The main task with Image Classification models is to produce a list of the most probable object categories present on an image, for example, to identify a tabby cat just after his dinner:



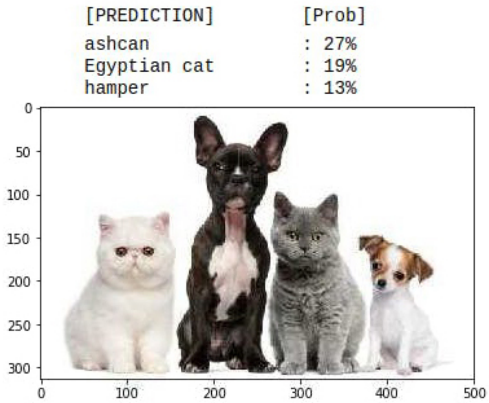
```
[PREDICTION]:
1) [tabby] ==> Probability of 30%
2) [bow tie] ==> Probability of 11%
3) [Egyptian cat] ==> Probability of 18%
```

But what happens when the cat jumps near the wine glass? The model still only recognizes the predominant category on the image, the tabby cat:



```
[PREDICTION]:

1) [tabby] ==> Probability of 53%
2) [tiger cat] ==> Probability of 23%
3) [Egyptian cat] ==> Probability of 10%
```

And what happens if there is not a dominant category on the image?



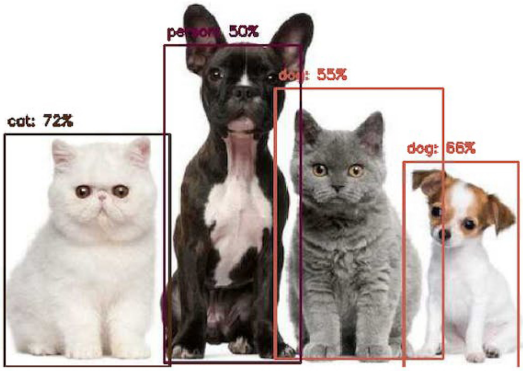
```
[PREDICTION]        [Prob]
ashcan              : 27%
Egyptian cat        : 19%
hamper              : 13%
```

The model identifies the above image utterly wrong as an "ashcan," possibly due to the color tonalities.

> The model used in all previous examples is MobileNet, which was trained with a large dataset, *ImageNet*.

To solve this issue, we need another type of model, where not only **multiple categories** (or labels) can be found but also **where** the objects are located on a given image.

As we can imagine, such models are much more complicated and bigger, for example, the **MobileNetV2 SSD FPN-Lite 320x320, trained with the COCO dataset.** This pre-trained object detection model is designed to locate up to 10 objects within an image, outputting a bounding box for each object detected. The below image is the result of such a model running on a Raspberry Pi:



Those models used for object detection (such as the MobileNet SSD or YOLO) usually have several MB in size, which is OK for Raspberry Pi but unsuitable for use with embedded devices, where the RAM is usually lower than 1 Mbyte.

An innovative solution for Object Detection: FOMO

Edge Impulse launched in 2022, **FOMO** (Faster Objects, More Objects), a novel solution for performing object detection on embedded devices, not only on the Nicla Vision (Cortex M7) but also on Cortex M4F CPUs (Arduino Nano33 and OpenMV M4 series) and the Espressif ESP32 devices (ESP-CAM and XIAO ESP32S3 Sense).

In this Hands-On lab, we will explore using FOMO with Object Detection, not entering many details about the model itself. To understand more about how the model works, you can go into the official FOMO announcement by Edge Impulse, where Louis Moreau and Mat Kelcey explain in detail how it works.

The Object Detection Project Goal

All Machine Learning projects need to start with a detailed goal. Let's assume we are in an industrial facility and must sort and count **wheels** and special **boxes**.

In other words, we should perform a multi-label classification, where each image can have three classes:

- Background (No objects)
- Box
- Wheel

Here are some not labeled image samples that we should use to detect the objects (wheels and boxes):

We are interested in which object is in the image, its location (centroid), and how many we can find on it. The object's size is not detected with FOMO, as with MobileNet SSD or YOLO, where the Bounding Box is one of the model outputs.

We will develop the project using the Nicla Vision for image capture and model inference. The ML project will be developed using the Edge Impulse Studio. But before starting the object detection project in the Studio, let's create a *raw dataset* (not labeled) with images that contain the objects to be detected.

Data Collection

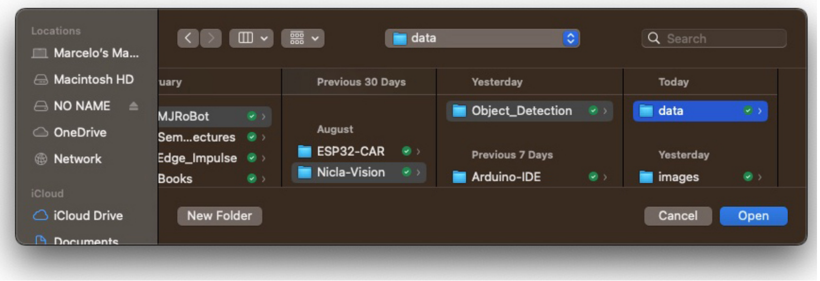
For image capturing, we can use:

- Web Serial Camera tool,
- Edge Impulse Studio,
- OpenMV IDE,
- A smartphone.

Here, we will use the **OpenMV IDE**.

Collecting Dataset with OpenMV IDE

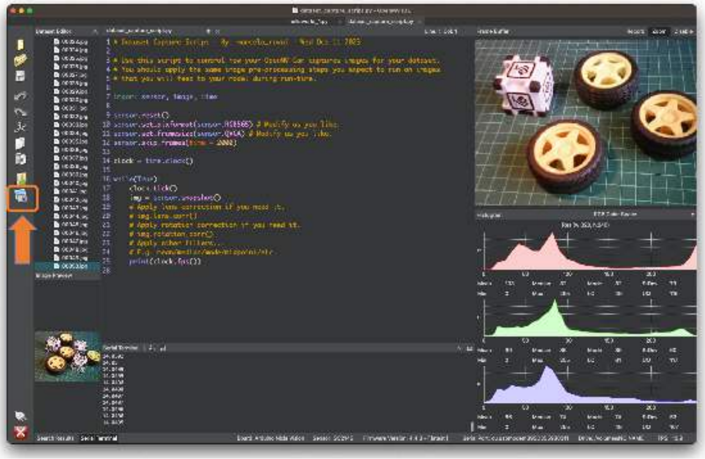
First, we create a folder on the computer where the data will be saved, for example, “data.” Next, on the OpenMV IDE, we go to Tools > Dataset Editor and select New Dataset to start the dataset collection:



Edge impulse suggests that the objects should be similar in size and not overlap for better performance. This is OK in an industrial facility, where the camera should be fixed, keeping the same distance from the objects to be detected. Despite that, we will also try using mixed sizes and positions to see the result.

> We will not create separate folders for our images because each contains multiple labels.

Connect the Nicla Vision to the OpenMV IDE and run the `dataset_capture_script.py`. Clicking on the Capture Image button will start capturing images:



```
# Dataset Capture Script - By: marcelo_rovai - Wed Oct 11 2023

# Use this script to control how your OpenMV Cam captures images for your dataset.
# You should apply the same image pre-processing steps you expect to run on images
# that you will feed to your model during run-time.

import sensor, image, time

sensor.reset()
sensor.set_pixformat(sensor.RGB565) # Modify as you like.
sensor.set_framesize(sensor.QVGA) # Modify as you like.
sensor.skip_frames(time = 2000)

clock = time.clock()

while(True):
    clock.tick()
    img = sensor.snapshot()
    # Apply lens correction if you need it.
    # img.lens_corr()
    # Apply rotation correction if you need it.
    # img.rotation_corr()
    # Apply other filters...
    # E.g. mean/median/mode/midpoint/etc.
    print(clock.fps())
```

We suggest using around 50 images to mix the objects and vary the number of each appearing on the scene. Try to capture different angles, backgrounds, and light conditions.

> The stored images use a QVGA frame size 320 × 240 and RGB565 (color pixel format).

After capturing your dataset, close the Dataset Editor Tool on the `Tools > Dataset Editor`.

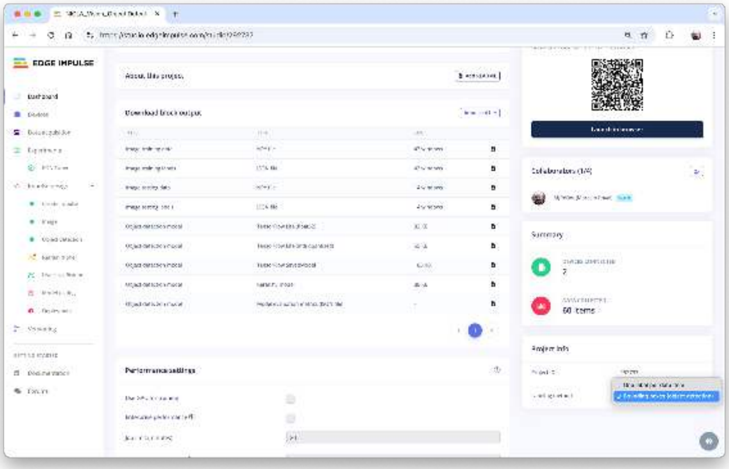
Edge Impulse Studio

Setup the project

Go to Edge Impulse Studio, enter your credentials at **Login** (or create an account), and start a new project.

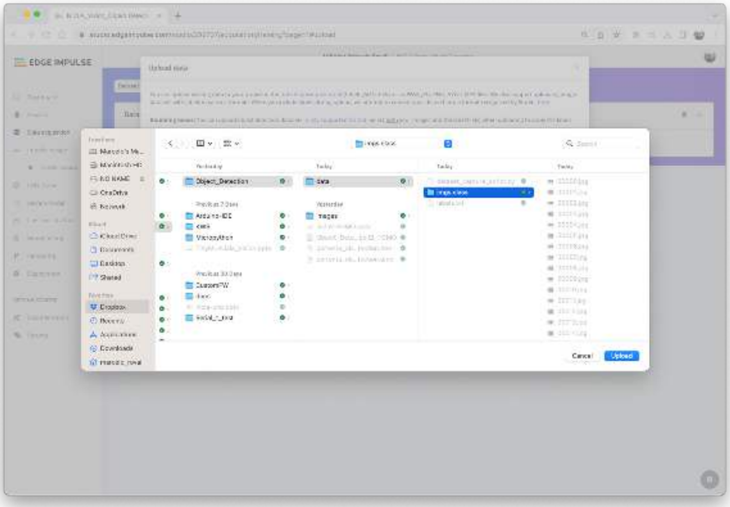
> Here, you can clone the project developed for this hands-on: NICLA_Vision_Object_Detection.

On the Project `Dashboard`, go to **Project info** and select **Bounding boxes (object detection),** and at the right-top of the page, select `Target`, **Arduino Nicla Vision (Cortex-M7)**.

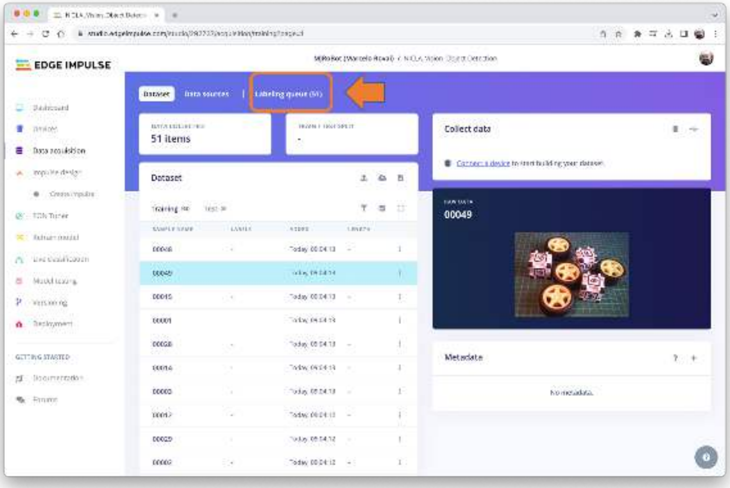


Uploading the unlabeled data

On Studio, go to the `Data acquisition` tab, and on the `UPLOAD DATA` section, upload from your computer files captured.



You can leave for the Studio to split your data automatically between Train and Test or do it manually.



All the unlabeled images (51) were uploaded, but they still need to be labeled appropriately before being used as a dataset in the project. The Studio has a tool for that purpose, which you can find in the link `Labeling queue (51)`.

There are two ways you can use to perform AI-assisted labeling on the Edge Impulse Studio (free version):

- Using yolov5

- Tracking objects between frames

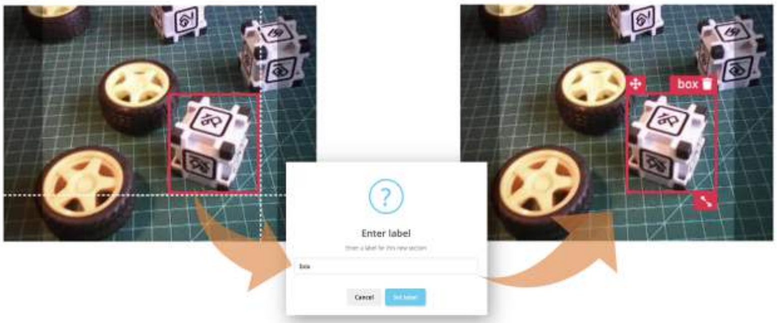
> Edge Impulse launched an auto-labeling feature for Enterprise customers, easing labeling tasks in object detection projects.

Ordinary objects can quickly be identified and labeled using an existing library of pre-trained object detection models from YOLOv5 (trained with the COCO dataset). But since, in our case, the objects are not part of COCO datasets, we should select the option of `tracking objects`. With this option, once you draw bounding boxes and label the images in one frame, the objects will be tracked automatically from frame to frame, *partially* labeling the new ones (not all are correctly labeled).

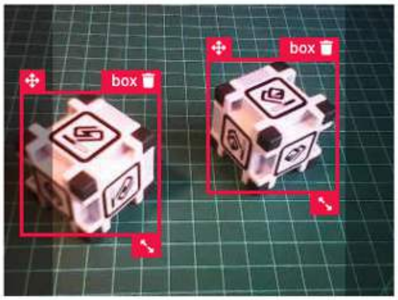
> If you already have a labeled dataset containing bounding boxes, import your data using the EI uploader.

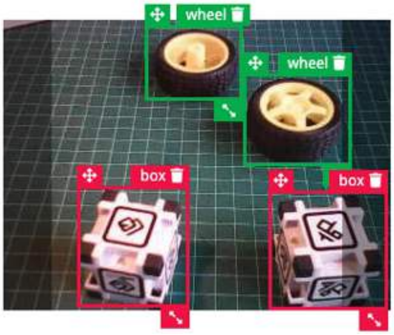
Labeling the Dataset

Starting with the first image of your unlabeled data, use your mouse to drag a box around an object to add a label. Then click **Save labels** to advance to the next item.

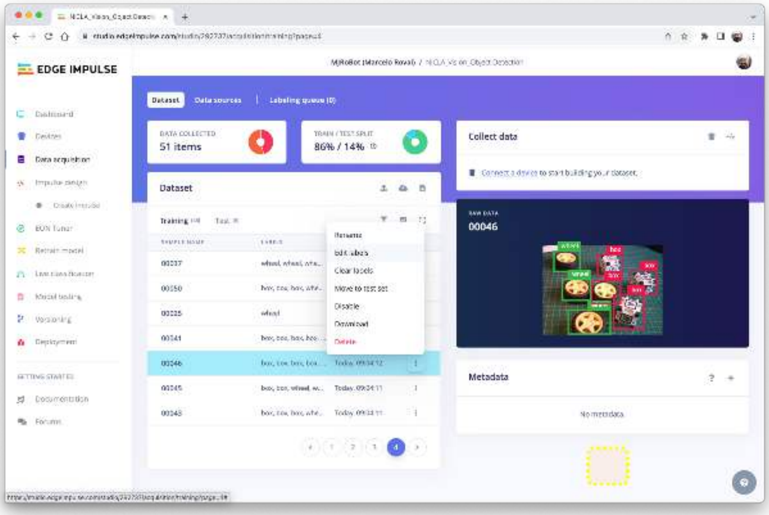


Continue with this process until the queue is empty. At the end, all images should have the objects labeled as those samples below:

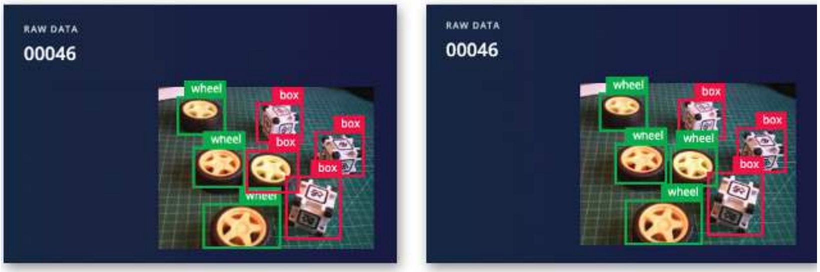




Next, review the labeled samples on the `Data acquisition` tab. If one of the labels is wrong, it can be edited using the *`three dots`* menu after the sample name:



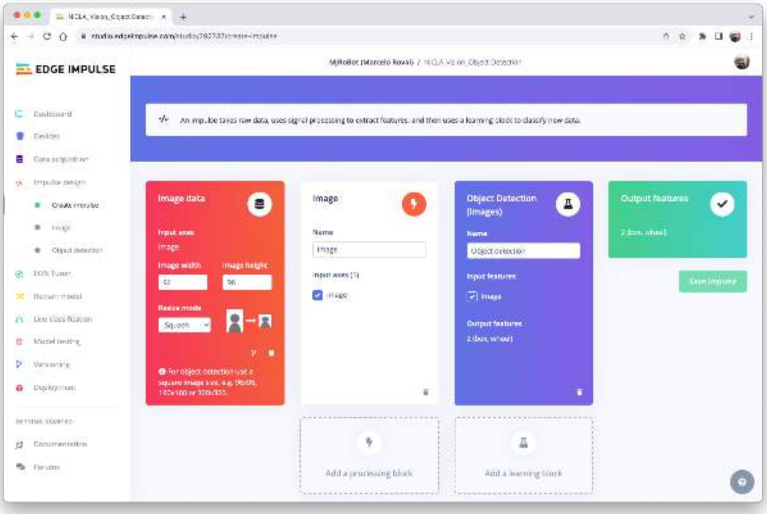
We will be guided to replace the wrong label and correct the dataset.



The Impulse Design

In this phase, we should define how to:

- **Pre-processing** consists of resizing the individual images from `320 x 240` to `96 x 96` and squashing them (squared form, without cropping). Afterward, the images are converted from RGB to Grayscale.
- **Design a Model,** in this case, "Object Detection."



An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

Image data
Input axes: image
Image width: 96
Image height: 96
Resize mode: Squash
For object detection use a square image size, e.g. 96x96, 160x160 or 320x320.

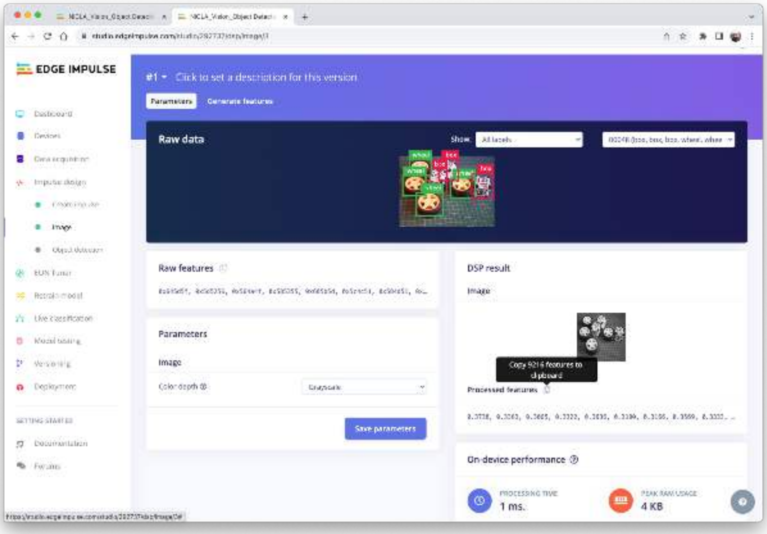
Image
Name: Image
Input axes (1): ☑ image

Object Detection (Images)
Name: Object detection
Input features: ☑ Image
Output features: 2 (box, wheel)

Output features
2 (box, wheel)

Preprocessing all dataset

In this section, select **Color depth** as `Grayscale`, suitable for use with FOMO models and Save `parameters`.



#1 Click to set a description for this version

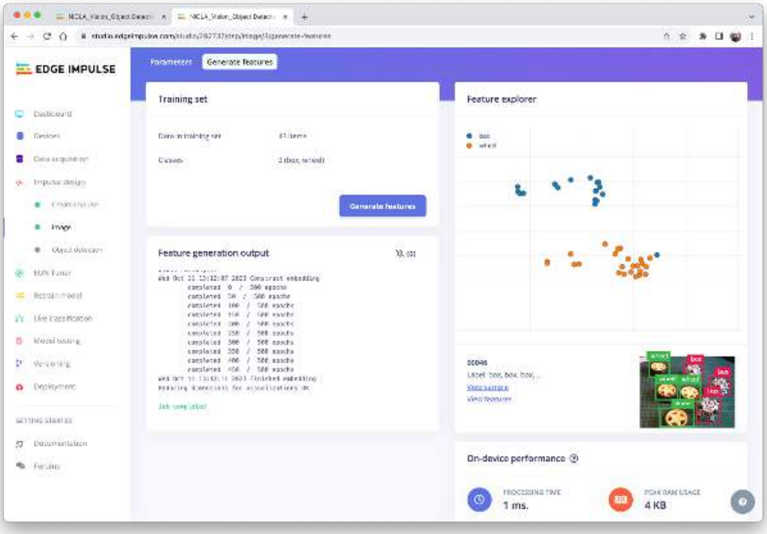
Raw features
0x5b5d5f, 0x5d5355, 0x5b4e4f, 0x5b5355, 0x6b5b5d, 0x5e5c5f, 0x5b4d5f, 0x...

Parameters
Image
Color depth: Grayscale

DSP result
Image
Processed features
0.3738, 0.3263, 0.3605, 0.3322, 0.3030, 0.3104, 0.3106, 0.3569, 0.3332...

On-device performance
PROCESSING TIME: 1 ms.
PEAK RAM USAGE: 4 KB

The Studio moves automatically to the next section, `Generate features`, where all samples will be pre-processed, resulting in a dataset with individual 96 × 96 × 1 images or 9,216 features.



Training set

Data in training set: [illegible] items

Classes: 2 (box, wheel)

Generate features

Feature generation output

```
Wed Oct 11 13:12:07 2023 Construct embedding
        completed  0  /  500 epochs
        completed  50  /  500 epochs
        completed  100  /  500 epochs
        completed  150  /  500 epochs
        completed  200  /  500 epochs
        completed  250  /  500 epochs
        completed  300  /  500 epochs
        completed  350  /  500 epochs
        completed  400  /  500 epochs
        completed  450  /  500 epochs
Wed Oct 11 13:12:11 2023 Finished embedding
Reducing dimensions for visualizations OK

Job completed
```

The feature explorer shows that all samples evidence a good separation after the feature generation.

> One of the samples (46) is apparently in the wrong space, but clicking on it confirms that the labeling is correct.

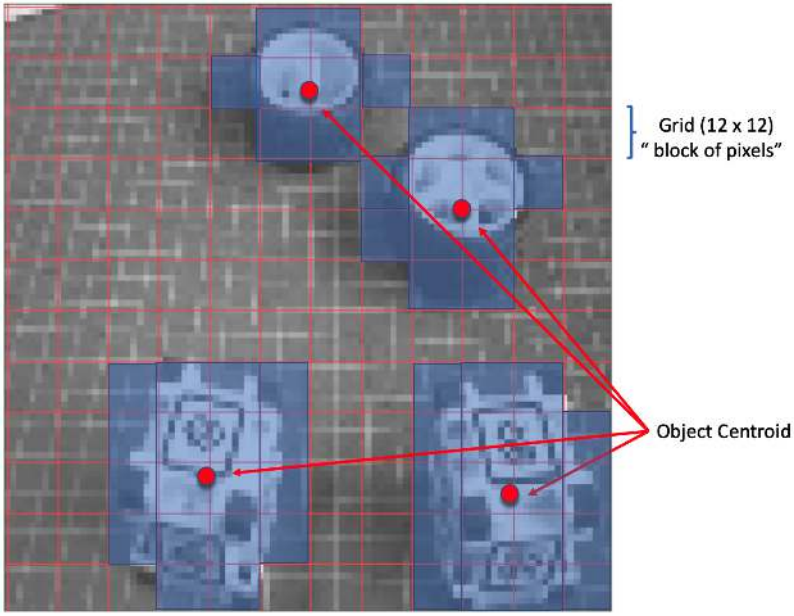
Model Design, Training, and Test

We will use FOMO, an object detection model based on MobileNetV2 (alpha 0.35) designed to coarsely segment an image into a grid of **background** vs **objects of interest** (here, *boxes* and *wheels*).

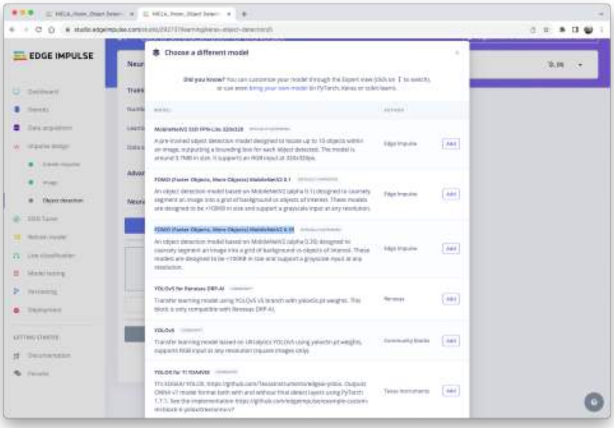
FOMO is an innovative machine learning model for object detection, which can use up to 30 times less energy and memory than traditional models like Mobilenet SSD and YOLOv5. FOMO can operate on microcontrollers with less than 200 KB of RAM. The main reason this is possible is that while other models calculate the object's size by drawing a square around it (bounding box), FOMO ignores the size of the image, providing only the information about where the object is located in the image, by means of its centroid coordinates.

How FOMO works?

FOMO takes the image in grayscale and divides it into blocks of pixels using a factor of 8. For the input of 96x96, the grid would be 12×12 $(96/8 = 12)$. Next, FOMO will run a classifier through each pixel block to calculate the probability that there is a box or a wheel in each of them and, subsequently, determine the regions that have the highest probability of containing the object (If a pixel block has no objects, it will be classified as *background*). From the overlap of the final region, the FOMO provides the coordinates (related to the image dimensions) of the centroid of this region.



For training, we should select a pre-trained model. Let's use the `FOMO (Faster Objects, More Objects) MobileNetV2 0.35`. This model uses around 250 KB of RAM and 80 KB of ROM (Flash), which suits well with our board since it has 1 MB of RAM and ROM.



Choose a different model

Did you know? You can customize your model through the Expert view (click on ⋮ to switch), or can even bring your own model in PyTorch, Keras or scikit-learn.

MobileNetV2 SSD FPN-Lite 320x320 — A pre-trained object detection model designed to locate up to 10 objects within an image, outputting a bounding box for each object detected. The model is around 3.7MB in size. It supports an RGB input at 320x320px. — Edge Impulse — Add

FOMO (Faster Objects, More Objects) MobileNetV2 0.1 — An object detection model based on MobileNetV2 (alpha 0.1) designed to coarsely segment an image into a grid of background vs objects of interest. These models are designed to be <100KB in size and support a grayscale input at any resolution. — Edge Impulse — Add

FOMO (Faster Objects, More Objects) MobileNetV2 0.35 — An object detection model based on MobileNetV2 (alpha 0.35) designed to coarsely segment an image into a grid of background vs objects of interest. These models are designed to be <100KB in size and support a grayscale input at any resolution. — Edge Impulse — Add

YOLOv5 for Renesas DRP-AI — Transfer learning model using YOLOv5 v5 branch with yolov5s.pt weights. This block is only compatible with Renesas DRP-AI. — Renesas — Add

YOLOv5 — Transfer learning model based on Ultralytics YOLOv5 using yolov5n.pt weights, supports RGB input at any resolution (square images only). — Community blocks — Add

YOLOX for TI TDA4VM — TI's EdgeAI YOLOX. https://github.com/TexasInstruments/edgeai-yolox. Outputs ONNX v7 model format both with and without final detect layers using PyTorch 1.7.1. See the implementation https://github.com/edgeimpulse/example-custom-ml-block-ti-yolox — Texas Instruments — Add

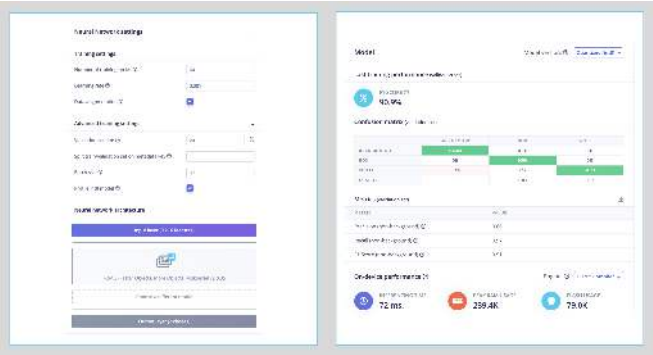
Regarding the training hyper-parameters, the model will be trained with:

- Epochs: 60,
- Batch size: 32
- Learning Rate: 0.001.

For validation during training, 20% of the dataset (*validation_dataset*) will be spared. For the remaining 80% (*train_dataset*), we will apply Data Augmentation, which will randomly flip, change the size and brightness of the image, and crop them, artificially increasing the number of samples on the dataset for training.

As a result, the model ends with an F1 score of around 91% (validation) and 93% (test data).

> Note that FOMO automatically added a 3rd label background to the two previously defined (*box* and *wheel*).



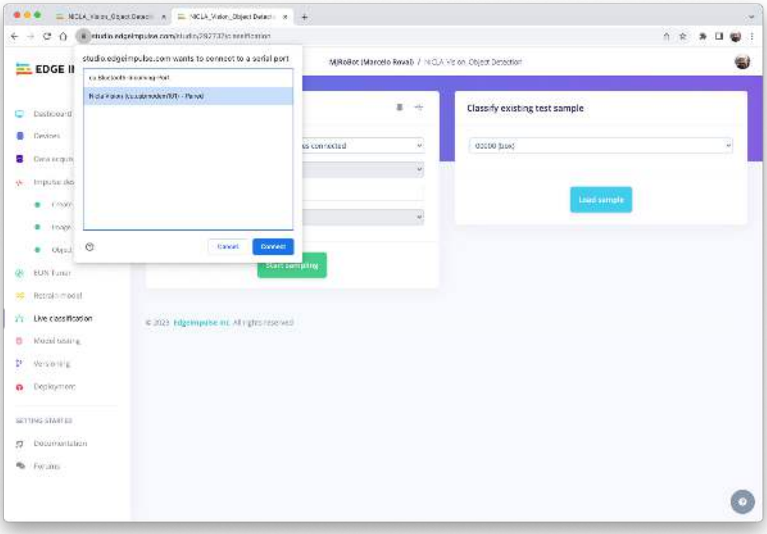
> In object detection tasks, accuracy is generally not the primary evaluation metric. Object detection involves classifying objects and providing bounding boxes around them, making it a more complex problem than simple classification. The issue is that we do not have the bounding box, only the centroids. In short, using accuracy as a metric could be misleading and may not provide a complete understanding of how well the model is performing. Because of that, we will use the F1 score.

Test model with "Live Classification"

Since Edge Impulse officially supports the Nicla Vision, let's connect it to the Studio. For that, follow the steps:

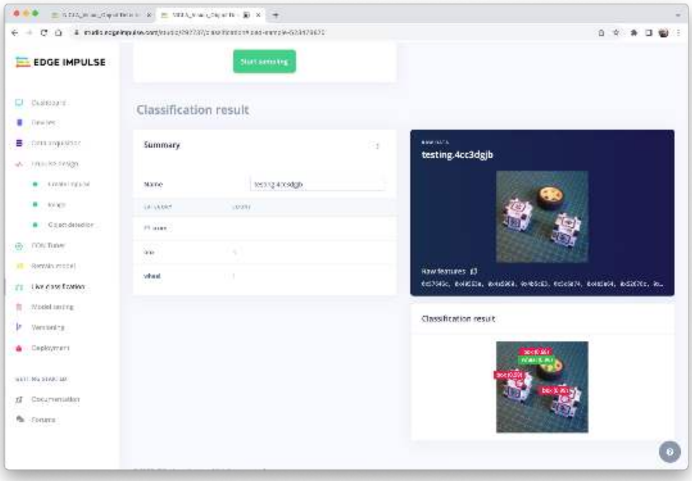
- Download the last EI Firmware and unzip it.
- Open the zip file on your computer and select the uploader related to your OS
- Put the Nicla-Vision on Boot Mode, pressing the reset button twice.
- Execute the specific batch code for your OS to upload the binary (`arduino-nicla-vision.bin`) to your board.

Go to `Live classification` section at EI Studio, and using *webUSB*, connect your Nicla Vision:



studio.edgeimpulse.com wants to connect to a serial port

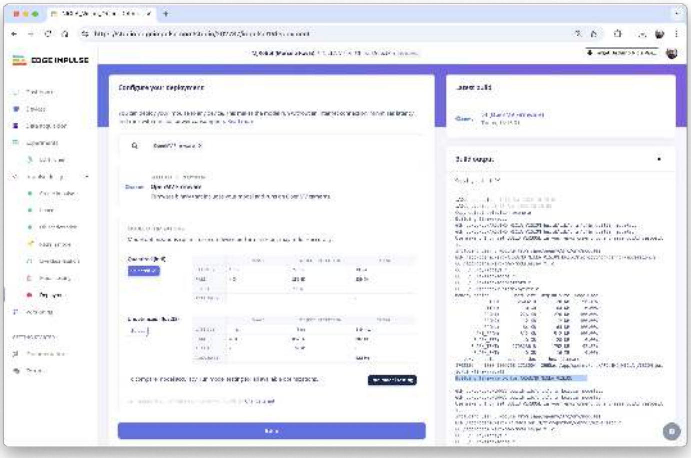
Once connected, you can use the Nicla to capture actual images to be tested by the trained model on Edge Impulse Studio.



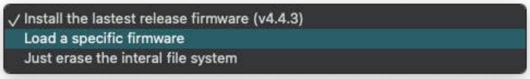
One thing to note is that the model can produce false positives and negatives. This can be minimized by defining a proper `Confidence Threshold` (use the `three dots` menu for the setup). Try with 0.8 or more.

Deploying the Model

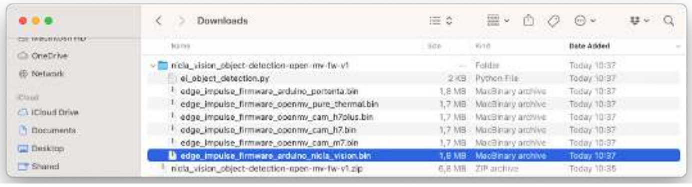
Select `OpenMV Firmware` on the Deploy Tab and press `[Build]`.



When you try to connect the Nicla with the OpenMV IDE again, it will try to update its FW. Choose the option `Load a specific firmware` instead. Or go to 'Tools > Runs Boatloader (Load Firmware).

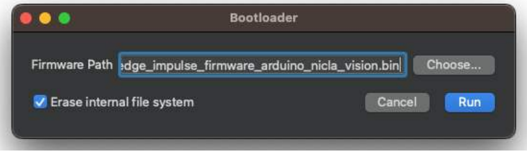

✓ Install the lastest release firmware (v4.4.3)
Load a specific firmware
Just erase the interal file system

You will find a ZIP file on your computer from the Studio. Open it:



Name	Size	Kind	Date Added
nicla_vision_object-detection-open-mv-fw-v1	--	Folder	Today 10:37
ei_object_detection.py	2 KB	Python File	Today 10:37
edge_impulse_firmware_arduino_portenta.bin	1,8 MB	MacBinary archive	Today 10:37
edge_impulse_firmware_openmv_pure_thermal.bin	1,7 MB	MacBinary archive	Today 10:37
edge_impulse_firmware_openmv_cam_h7plus.bin	1,7 MB	MacBinary archive	Today 10:37
edge_impulse_firmware_openmv_cam_h7.bin	1,7 MB	MacBinary archive	Today 10:37
edge_impulse_firmware_openmv_cam_m7.bin	1,7 MB	MacBinary archive	Today 10:37
edge_impulse_firmware_arduino_nicla_vision.bin	1,8 MB	MacBinary archive	Today 10:37
nicla_vision_object-detection-open-mv-fw-v1.zip	6,8 MB	ZIP archive	Today 10:36

Load the .bin file to your board:


Bootloader

Firmware Path: edge_impulse_firmware_arduino_nicla_vision.bin

☑ Erase internal file system

After the download is finished, a pop-up message will be displayed. `Press OK`, and open the script **ei_object_detection.py** downloaded from the Studio.

> Note: If a Pop-up appears saying that the FW is out of date, press `[NO]`, to upgrade it.

Before running the script, let's change a few lines. Note that you can leave the window definition as 240 × 240 and the camera capturing images as QVGA/RGB. The captured image will be pre-processed by the FW deployed from Edge Impulse

```
import sensor
import time
import ml
from ml.utils import NMS
import math
import image

sensor.reset()  # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)  # Set pixel format (RGB565or GRAYSCALE)
sensor.set_framesize(sensor.QVGA)  # Set frame size to QVGA (320x240)
sensor.skip_frames(time=2000)  # Let the camera adjust.
```

Redefine the minimum confidence, for example, to 0.8 to minimize false positives and negatives.

```
min_confidence = 0.8
```

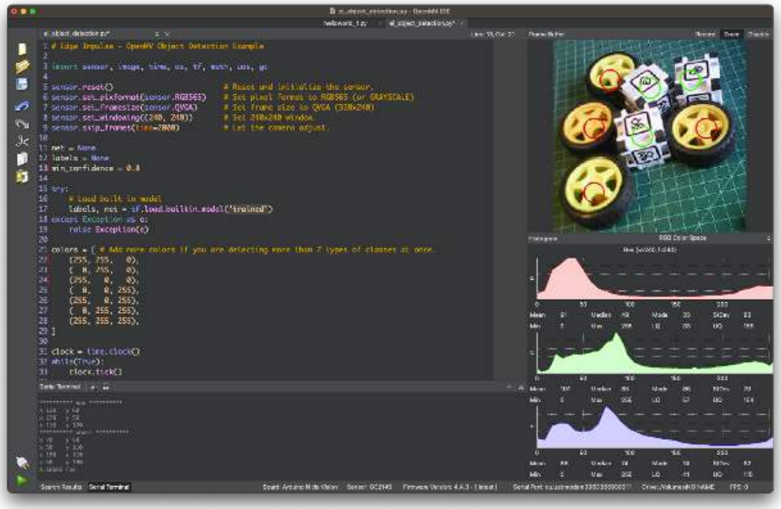
Change if necessary, the color of the circles that will be used to display the detected object's centroid for a better contrast.

```
threshold_list = [(math.ceil(min_confidence * 255), 255)]

# Load built-in model
model = ml.Model("trained")
print(model)

# Alternatively, models can be loaded from the filesystem storage.
# model = ml.Model('<object_detection_modelwork>.tflite', load_to_fb=True)
# labels = [line.rstrip('\n') for line in open("labels.txt")]

colors = [ # Add more colors if you are detecting more
           # than 7 types of classes at once.
    (255, 255,   0), # background: yellow (not used)
    (  0, 255,   0), # cube: green
    (255,   0,   0), # wheel: red
    (  0,   0, 255), # not used
    (255,   0, 255), # not used
    (  0, 255, 255), # not used
    (255, 255, 255), # not used
]
```

Keep the remaining code as it is

```
# FOMO outputs an image per class where each pixel in the image is the centroid
# object. So, we will get those output images and then run find_blobs() on them
# centroids. We will also run get_stats() on the detected blobs to determine the
# The Non-Max-Supression (NMS) object then filters out overlapping detections ar
# position in the output image back to the original input image. The function th
# list per class which each contain a list of (rect, score) tuples representing
# objects.

def fomo_post_process(model, inputs, outputs):
    n, oh, ow, oc = model.output_shape[0]
    nms = NMS(ow, oh, inputs[0].roi)
    for i in range(oc):
        img = image.Image(outputs[0][0, :, :, i] * 255)
        blobs = img.find_blobs(
            threshold_list, x_stride=1, area_threshold=1, pixels_threshold=1
        )
        for b in blobs:
            rect = b.rect()
            x, y, w, h = rect
            score = (
                img.get_statistics(thresholds=threshold_list, roi=rect).l_mean(
            )
            nms.add_bounding_box(x, y, x + w, y + h, score, i)
    return nms.get_bounding_boxes()


clock = time.clock()
while True:
    clock.tick()

    img = sensor.snapshot()

    for i, detection_list in enumerate(model.predict([img], callback=fomo_post_p
        if i == 0:
            continue  # background class
        if len(detection_list) == 0:
            continue  # no detections for this class?

        print("********** %s **********" % model.labels[i])
        for (x, y, w, h), score in detection_list:
            center_x = math.floor(x + (w / 2))
            center_y = math.floor(y + (h / 2))
            print(f"x {center_x}\ty {center_y}\tscore {score}")
            img.draw_circle((center_x, center_y, 12), color=colors[i])

    print(clock.fps(), "fps", end="\n")
```

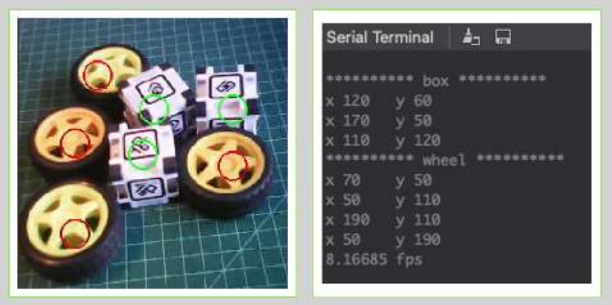
and press the `green Play button` to run the code:



```
# Edge Impulse - OpenMV Object Detection Example

import sensor, image, time, os, tf, math, uos, gc

sensor.reset()                         # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)    # Set pixel format to RGB565 (or GRAYSCALE)
sensor.set_framesize(sensor.QVGA)      # Set frame size to QVGA (320x240)
sensor.set_windowing((240, 240))       # Set 240x240 window.
sensor.skip_frames(time=2000)          # Let the camera adjust.

net = None
labels = None
min_confidence = 0.8

try:
    # Load built in model
    labels, net = tf.load_builtin_model('trained')
except Exception as e:
    raise Exception(e)

colors = [ # Add more colors if you are detecting more than 7 types of classes at once.
    (255, 255,   0),
    (  0, 255,   0),
    (255,   0,   0),
    (  0,   0, 255),
    (255,   0, 255),
    (  0, 255, 255),
    (255, 255, 255),
]

clock = time.clock()
while(True):
    clock.tick()
```

From the camera's view, we can see the objects with their centroids marked with 12 pixel-fixed circles (each circle has a distinct color, depending on its class). On the Serial Terminal, the model shows the labels detected and their position on the image window (240 × 240).

Be aware that the coordinate origin is in the upper left corner.



```
********** box **********
x 120   y 60
x 170   y 50
x 110   y 120
********** wheel **********
x 70    y 50
x 50    y 110
x 190   y 110
x 50    y 190
8.16685 fps
```

Note that the frames per second rate is around 8 fps (similar to what we got with the Image Classification project). This happens because FOMO is cleverly built over a CNN model, not with an object detection model like the SSD MobileNet or YOLO. For example, when running a MobileNetV2 SSD FPN-Lite 320 × 320 model on a Raspberry Pi 4, the latency is around 5 times higher (around 1.5 fps)

Here is a short video showing the inference results: https://youtu.be/JbpoqRp3BbM

Conclusion

FOMO is a significant leap in the image processing space, as Louis Moreau and Mat Kelcey put it during its launch in 2022:

> FOMO is a ground-breaking algorithm that brings real-time object detection, tracking, and counting to microcontrollers for the first time.

Multiple possibilities exist for exploring object detection (and, more precisely, counting them) on embedded devices. This can be very useful on projects counting bees, for example.

Resources

- Edge Impulse Project

Keyword Spotting (KWS)

Figure 20.6: *DALL·E 3 Prompt: 1950s style cartoon scene set in a vintage audio research room. Two Afro-American female scientists are at the center. One holds a magnifying glass, closely examining ancient circuitry, while the other takes notes. On their wooden table, there are multiple boards with sensors, notably featuring a microphone. Behind these boards, a computer with a large, rounded back displays the Arduino IDE. The IDE showcases code for LED pin assignments and machine learning inference for voice command detection. A distinct window in the IDE, the Serial Monitor, reveals outputs indicating the spoken commands 'yes' and 'no'. The room ambiance is nostalgic with vintage lamps, classic audio analysis tools, and charts depicting FFT graphs and time-domain curves.*

Overview

Having already explored the Nicla Vision board in the *Image Classification* and *Object Detection* applications, we are now shifting our focus to voice-activated applications with a project on Keyword Spotting (KWS).

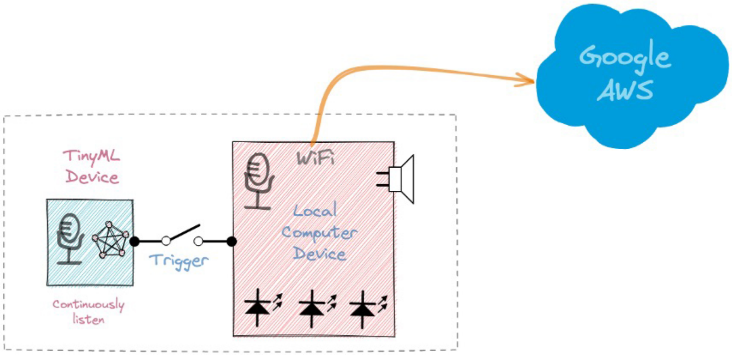
As introduced in the *Feature Engineering for Audio Classification* Hands-On tutorial, Keyword Spotting (KWS) is integrated into many voice recognition systems, enabling devices to respond to specific words or phrases. While this technology underpins popular devices like Google Assistant or Amazon Alexa, it's equally applicable and feasible on smaller, low-power devices. This tutorial will guide you through implementing a KWS system using TinyML on the Nicla Vision development board equipped with a digital microphone.

Our model will be designed to recognize keywords that can trigger device wake-up or specific actions, bringing them to life with voice-activated commands.

How does a voice assistant work?

As said, *voice assistants* on the market, like Google Home or Amazon Echo-Dot, only react to humans when they are "waked up" by particular keywords such as " Hey Google" on the first one and "Alexa" on the second.

In other words, recognizing voice commands is based on a multi-stage model or Cascade Detection.



Stage 1: A small microprocessor inside the Echo Dot or Google Home continuously listens, waiting for the keyword to be spotted, using a TinyML model at the edge (KWS application).

Stage 2: Only when triggered by the KWS application on Stage 1 is the data sent to the cloud and processed on a larger model.

The video below shows an example of a Google Assistant being programmed on a Raspberry Pi (Stage 2), with an Arduino Nano 33 BLE as the TinyML device (Stage 1).

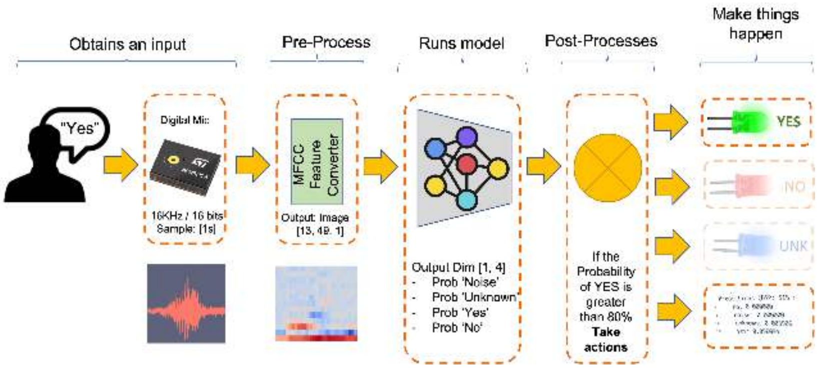
https://youtu.be/e_OPgcnsyvM

> To explore the above Google Assistant project, please see the tutorial: Building an Intelligent Voice Assistant From Scratch.

In this KWS project, we will focus on Stage 1 (KWS or Keyword Spotting), where we will use the Nicla Vision, which has a digital microphone that will be used to spot the keyword.

The KWS Hands-On Project

The diagram below gives an idea of how the final KWS application should work (during inference):



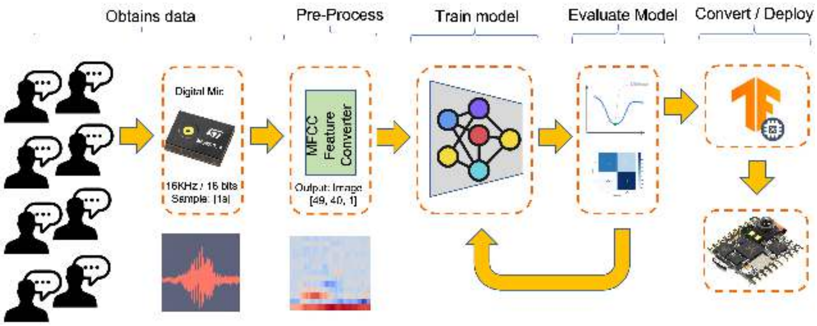
Our KWS application will recognize four classes of sound:

- **YES** (Keyword 1)
- **NO** (Keyword 2)
- **NOISE** (no words spoken; only background noise is present)
- **UNKNOWN** (a mix of different words than YES and NO)

> For real-world projects, it is always advisable to include other sounds besides the keywords, such as "Noise" (or Background) and "Unknown."

The Machine Learning workflow

The main component of the KWS application is its model. So, we must train such a model with our specific keywords, noise, and other words (the "unknown"):



Dataset

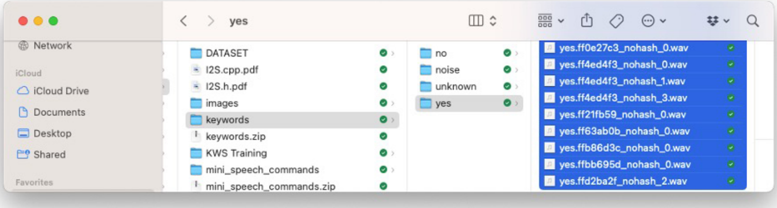
The critical component of any Machine Learning Workflow is the **dataset**. Once we have decided on specific keywords, in our case (*YES* and NO), we can take advantage of the dataset developed by Pete Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition." This dataset has 35 keywords (with +1,000 samples each), such as yes, no, stop, and go. In words such as *yes* and *no*, we can get 1,500 samples.

You can download a small portion of the dataset from Edge Studio (Keyword spotting pre-built dataset), which includes samples from the four classes we will use in this project: yes, no, noise, and background. For this, follow the steps below:

- Download the keywords dataset.
- Unzip the file to a location of your choice.

Uploading the dataset to the Edge Impulse Studio

Initiate a new project at Edge Impulse Studio (EIS) and select the `Upload Existing Data` tool in the `Data Acquisition` section. Choose the files to be uploaded:



yes

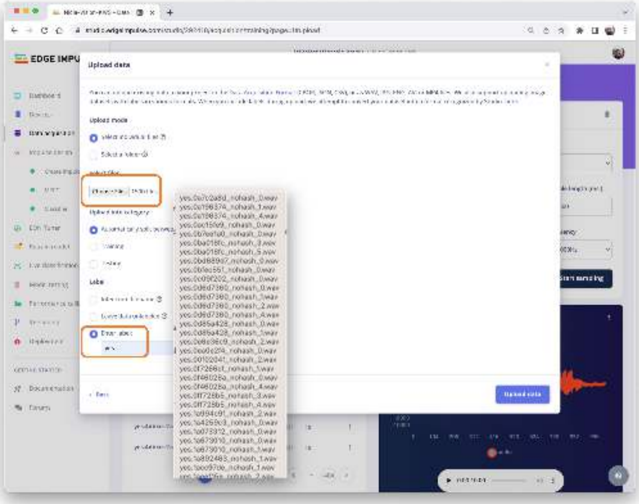
Network
iCloud
iCloud Drive
Documents
Desktop
Shared
Favorites

DATASET
I2S.cpp.pdf
I2S.h.pdf
images
keywords
keywords.zip
KWS Training
mini_speech_commands
mini_speech_commands.zip

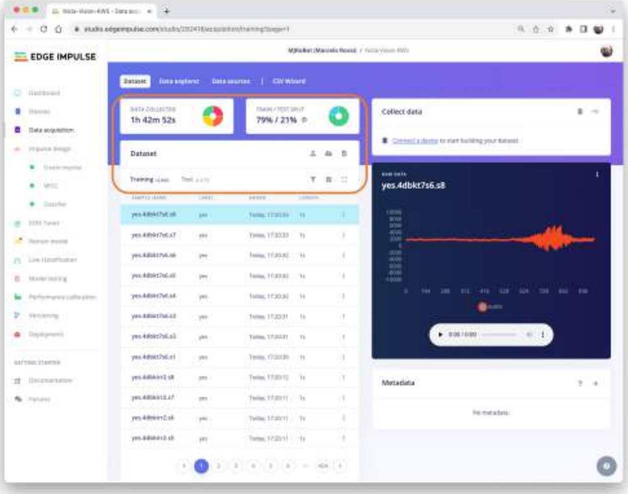
no
noise
unknown
yes

yes.ff0e27c3_nohash_0.wav
yes.ff4ed4f3_nohash_0.wav
yes.ff4ed4f3_nohash_1.wav
yes.ff4ed4f3_nohash_3.wav
yes.ff21fb59_nohash_0.wav
yes.ff63ab0b_nohash_0.wav
yes.ffb86d3c_nohash_0.wav
yes.ffbb695d_nohash_0.wav
yes.ffd2ba2f_nohash_2.wav

Define the Label, select `Automatically split between train and test`, and `Upload data` to the EIS. Repeat for all classes.



The dataset will now appear in the `Data acquisition` section. Note that the approximately 6,000 samples (1,500 for each class) are split into Train (4,800) and Test (1,200) sets.



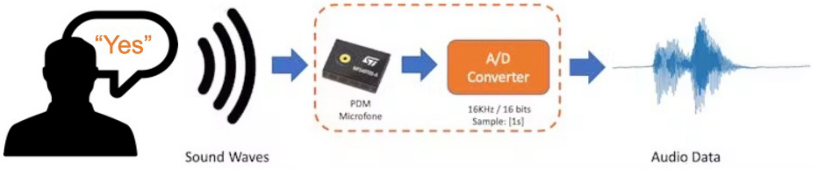
Capturing additional Audio Data

Although we have a lot of data from Pete's dataset, collecting some words spoken by us is advised. When working with accelerometers, creating a dataset with data captured by the same type of sensor is essential. In the case of *sound*, this is optional because what we will classify is, in reality, *audio* data.

> The key difference between sound and audio is the type of energy. Sound is mechanical perturbation (longitudinal sound waves) that propagate through a medium, causing variations of pressure in it. Audio is an electrical (analog or digital) signal representing sound.

When we pronounce a keyword, the sound waves should be converted to audio data. The conversion should be done by sampling the signal generated by the microphone at a 16 KHz frequency with 16-bit per sample amplitude.

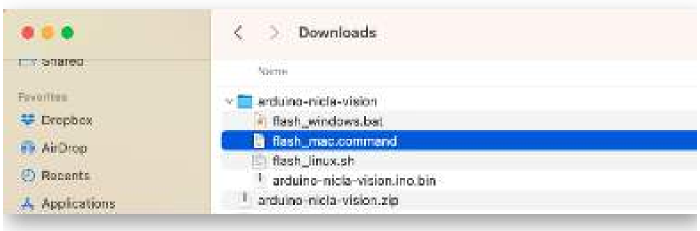
So, any device that can generate audio data with this basic specification (16 KHz/16 bits) will work fine. As a *device*, we can use the NiclaV, a computer, or even your mobile phone.



Using the NiclaV and the Edge Impulse Studio

As we learned in the chapter *Setup Nicla Vision*, EIS officially supports the Nicla Vision, which simplifies the capture of the data from its sensors, including the microphone. So, please create a new project on EIS and connect the Nicla to it, following these steps:

- Download the last updated EIS Firmware and unzip it.
- Open the zip file on your computer and select the uploader corresponding to your OS:

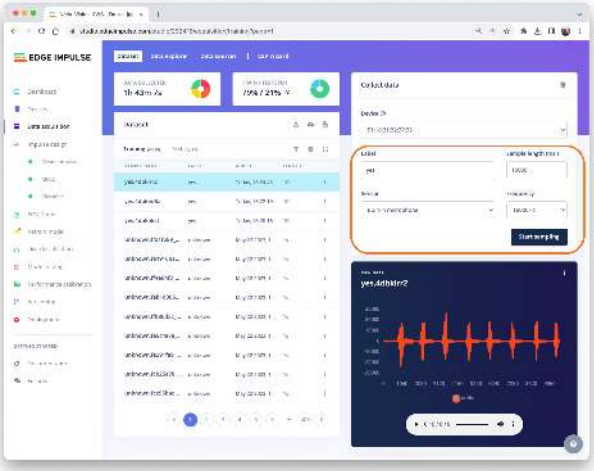


arduino-nicla-vision
flash_windows.bat
flash_mac.command
flash_linux.sh
arduino-nicla-vision.ino.bin
arduino-nicla-vision.zip

- Put the NiclaV in Boot Mode by pressing the reset button twice.
- Upload the binary *arduino-nicla-vision.bin* to your board by running the batch code corresponding to your OS.

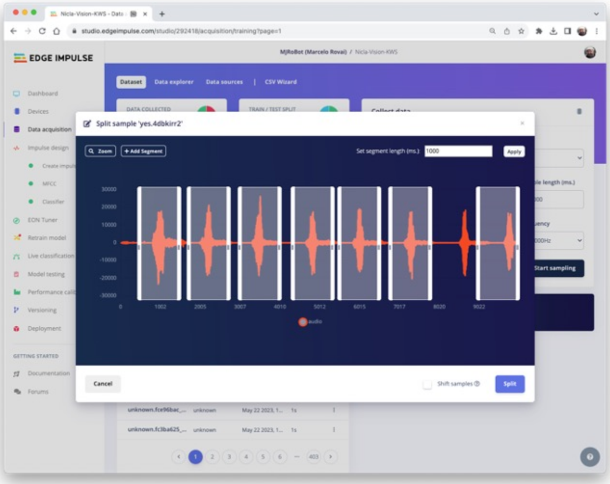
Go to your project on EIS, and on the `Data Acquisition tab`, select `WebUSB`. A window will pop up; choose the option that shows that the `Nicla is paired` and press `[Connect]`.

You can choose which sensor data to pick in the `Collect Data` section on the `Data Acquisition` tab. Select: `Built-in microphone`, define your `label` (for example, *yes*), the sampling `Frequency`[16000Hz], and the `Sample length (in milliseconds)`, for example [10s]. `Start sampling`.



Data on Pete’s dataset have a length of 1s, but the recorded samples are 10s long and must be split into 1s samples. Click on `three dots` after the sample name and select `Split sample`.

A window will pop up with the Split tool.

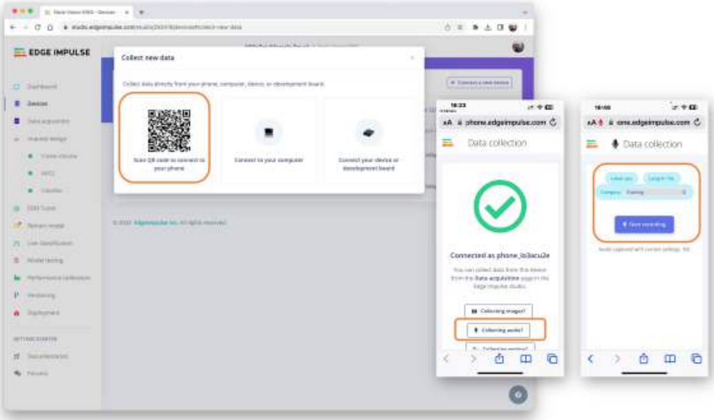


Once inside the tool, split the data into 1-second (1000 ms) records. If necessary, add or remove segments. This procedure should be repeated for all new samples.

Using a smartphone and the EI Studio

You can also use your PC or smartphone to capture audio data, using a sampling frequency of 16 KHz and a bit depth of 16.

Go to `Devices`, scan the `QR Code` using your phone, and click on the link. A data Collection app will appear in your browser. Select `Collecting Audio`, and define your `Label`, data capture `Length`, and `Category`.



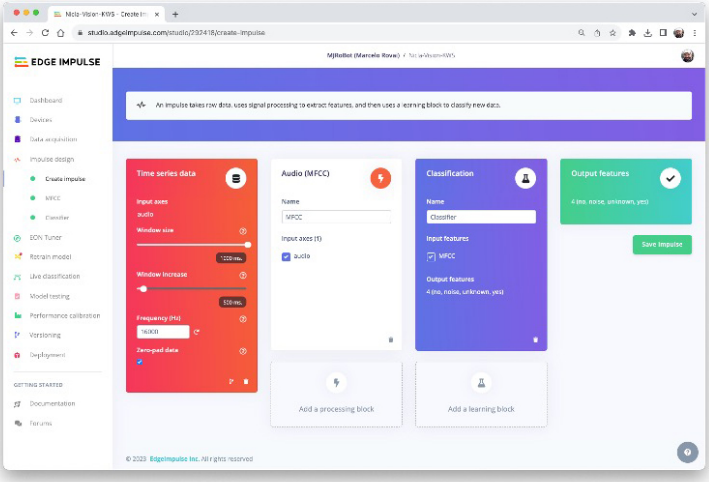
Repeat the same procedure used with the NiclaV.

> Note that any app, such as Audacity, can be used for audio recording, provided you use 16 KHz/16-bit depth samples.

Creating Impulse (Pre-Process / Model definition)

An **impulse** *takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.*

Impulse Design



An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

Time series data
Input axes: audio
Window size: 1000 ms.
Window increase: 500 ms.
Frequency (Hz): 16000
Zero-pad data: ☑

Audio (MFCC)
Name: MFCC
Input axes (1): ☑ audio

Classification
Name: Classifier
Input features: ☑ MFCC
Output features: 4 (no, noise, unknown, yes)

Output features
4 (no, noise, unknown, yes)

First, we will take the data points with a 1-second window, augmenting the data and sliding that window in 500 ms intervals. Note that the option zero-pad data is set. It is essential to fill with 'zeros' samples smaller than 1 second (in some cases, some samples can result smaller than the 1000 ms window on the split tool to avoid noise and spikes).

Each 1-second audio sample should be pre-processed and converted to an image (for example, 13 × 49 × 1). As discussed in the *Feature Engineering for Audio Classification* Hands-On tutorial, we will use `Audio (MFCC)`, which extracts features from audio signals using Mel Frequency Cepstral Coefficients, which are well suited for the human voice, our case here.

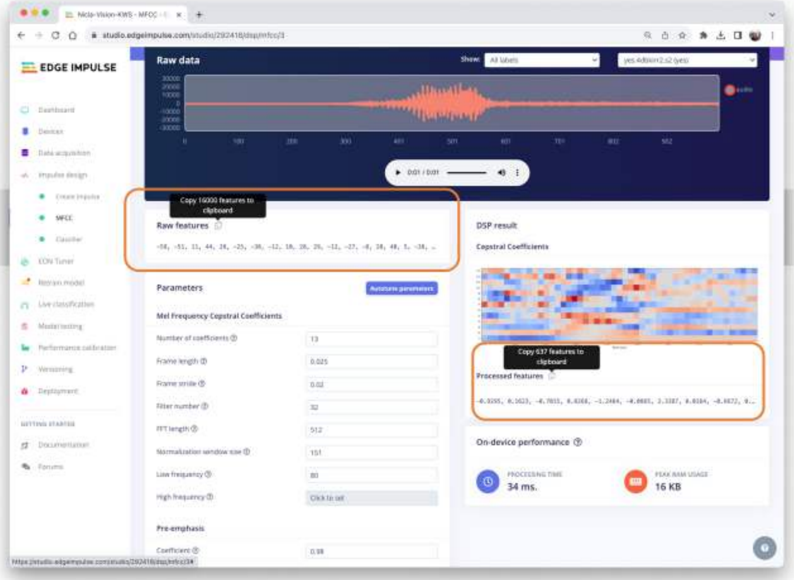
Next, we select the `Classification` block to build our model from scratch using a Convolution Neural Network (CNN).

> Alternatively, you can use the `Transfer Learning (Keyword Spotting)` block, which fine-tunes a pre-trained keyword spotting model on your data. This approach has good performance with relatively small keyword datasets.

Pre-Processing (MFCC)

The following step is to create the features to be trained in the next phase:

We could keep the default parameter values, but we will use the DSP `Autotune parameters` option.



Raw features

Parameters

Mel Frequency Cepstral Coefficients

Number of coefficients: 13
Frame length: 0.025
Frame stride: 0.02
Filter number: 32
FFT length: 512
Normalization window size: 151
Low frequency: 80
High frequency: Click to set

Pre-emphasis

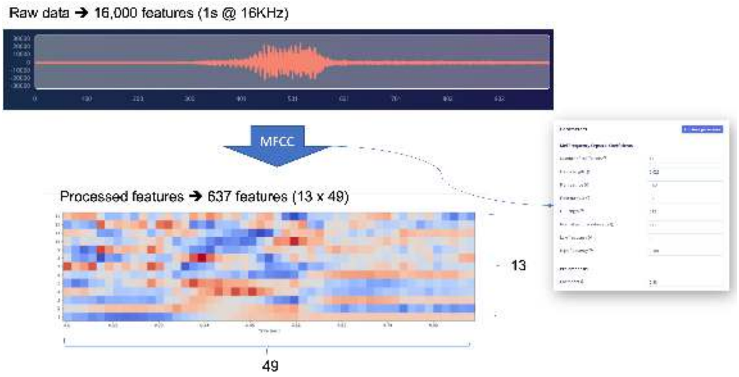
Coefficient: 0.98

Processed features

On-device performance

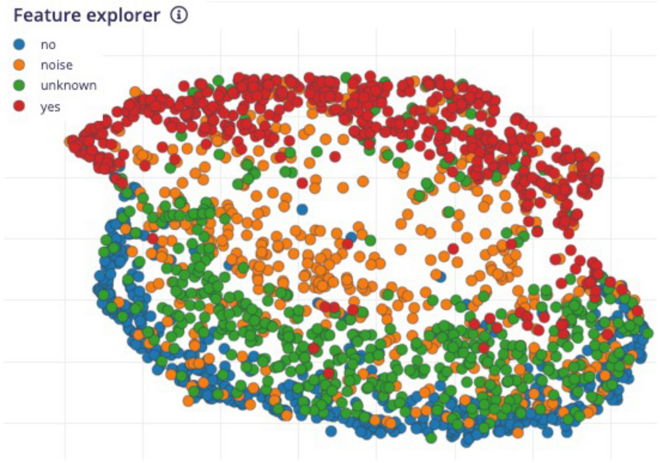
PROCESSING TIME: 34 ms.
PEAK RAM USAGE: 16 KB

We will take the `Raw features` (our 1-second, 16 KHz sampled audio data) and use the MFCC processing block to calculate the `Processed features`. For every 16,000 raw features (16,000 × 1 second), we will get 637 processed features (13 × 49).


Raw data ➔ 16,000 features (1s @ 16KHz)
MFCC
Processed features ➔ 637 features (13 x 49)
13
49

The result shows that we only used a small amount of memory to pre-process data (16 KB) and a latency of 34 ms, which is excellent. For example, on an Arduino Nano (Cortex-M4f @ 64 MHz), the same pre-process will take around 480 ms. The parameters chosen, such as the `FFT length` [512], will significantly impact the latency.

Now, let's `Save parameters` and move to the `Generated features` tab, where the actual features will be generated. Using UMAP, a dimension reduction technique, the `Feature explorer` shows how the features are distributed on a two-dimensional plot.



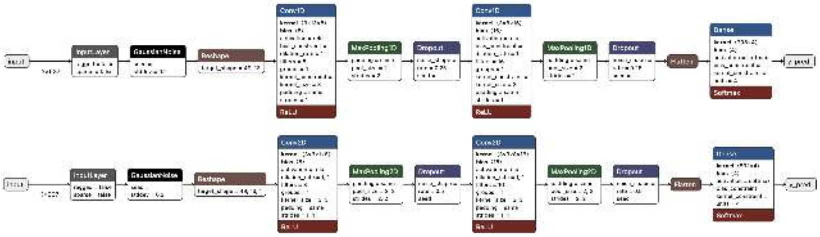
The result seems OK, with a visually clear separation between *yes* features (in red) and *no* features (in blue). The *unknown* features seem nearer to the *no space* than the *yes*. This suggests that the keyword *no* has more propensity to false positives.

Going under the hood

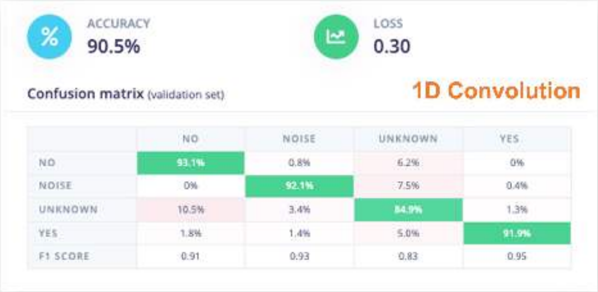
To understand better how the raw sound is preprocessed, look at the *Feature Engineering for Audio Classification* chapter. You can play with the MFCC features generation by downloading this notebook from GitHub or [Opening it In Colab]

Model Design and Training

We will use a simple Convolution Neural Network (CNN) model, tested with 1D and 2D convolutions. The basic architecture has two blocks of Convolution + MaxPooling ([8] and [16] filters, respectively) and a Dropout of [0.25] for the 1D and [0.5] for the 2D. For the last layer, after Flattening, we have [4] neurons, one for each class:



As hyper-parameters, we will have a `Learning Rate` of [0.005] and a model trained by [100] epochs. We will also include a data augmentation method based on SpecAugment. We trained the 1D and the 2D models with the same hyperparameters. The 1D architecture had a better overall result (90.5% accuracy when compared with 88% of the 2D, so we will use the 1D.

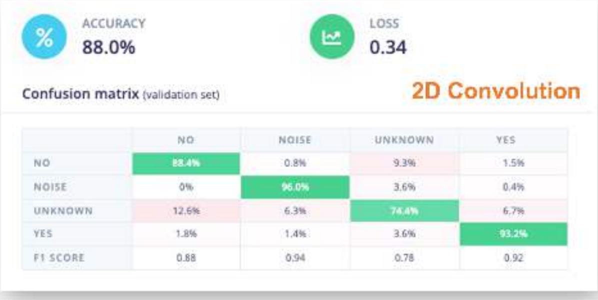

ACCURACY 90.5%

LOSS 0.30

Confusion matrix (validation set)

1D Convolution

	NO	NOISE	UNKNOWN	YES
NO	93.1%	0.8%	6.2%	0%
NOISE	0%	92.1%	7.5%	0.4%
UNKNOWN	10.5%	3.4%	84.9%	1.3%
YES	1.8%	1.4%	5.0%	91.9%
F1 SCORE	0.91	0.93	0.83	0.95


ACCURACY 88.0%

LOSS 0.34

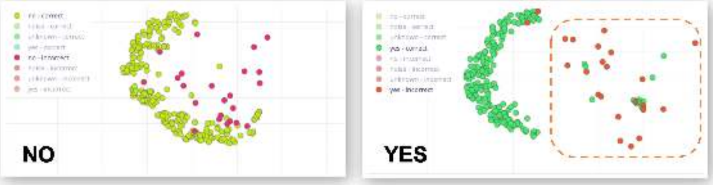
Confusion matrix (validation set)

2D Convolution

	NO	NOISE	UNKNOWN	YES
NO	88.4%	0.8%	9.3%	1.5%
NOISE	0%	96.0%	3.6%	0.4%
UNKNOWN	12.6%	6.3%	74.4%	6.7%
YES	1.8%	1.4%	3.6%	93.2%
F1 SCORE	0.88	0.94	0.78	0.92

> Using 1D convolutions is more efficient because it requires fewer parameters than 2D convolutions, making them more suitable for resource-constrained environments.

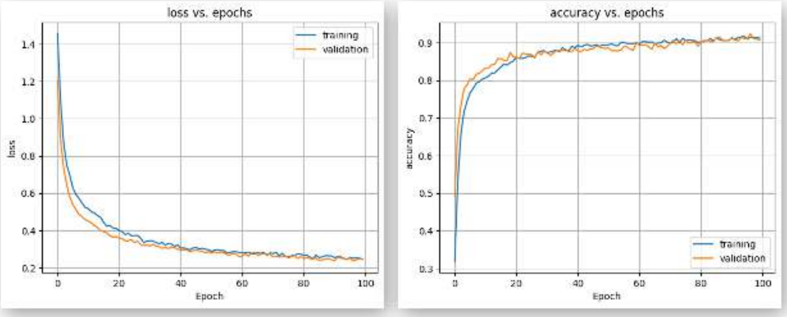
It is also interesting to pay attention to the 1D Confusion Matrix. The F1 Score for `yes` is 95%, and for `no`, 91%. That was expected by what we saw with the Feature Explorer (`no` and `unknown` at close distance). In trying to improve the result, you can inspect closely the results of the samples with an error.



Listen to the samples that went wrong. For example, for `yes`, most of the mistakes were related to a yes pronounced as "yeh". You can acquire additional samples and then retrain your model.

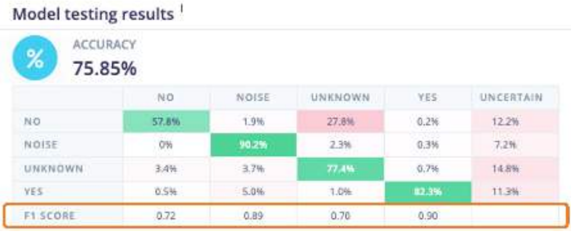
Going under the hood

If you want to understand what is happening "under the hood," you can download the pre-processed dataset (`MFCC training data`) from the `Dashboard` tab and run this Jupyter Notebook, playing with the code or [Opening it In Colab]. For example, you can analyze the accuracy by each epoch:



Testing

Testing the model with the data reserved for training (Test Data), we got an accuracy of approximately 76%.



Model testing results

ACCURACY
75.85%

	NO	NOISE	UNKNOWN	YES	UNCERTAIN
NO	57.8%	1.9%	27.8%	0.2%	12.2%
NOISE	0%	90.2%	2.3%	0.3%	7.2%
UNKNOWN	3.4%	3.7%	77.4%	0.7%	14.8%
YES	0.5%	5.0%	1.0%	82.3%	11.3%
F1 SCORE	0.72	0.89	0.70	0.90	

Inspecting the F1 score, we can see that for YES, we got 0.90, an excellent result since we expect to use this keyword as the primary "trigger" for our KWS project. The worst result (0.70) is for UNKNOWN, which is OK.

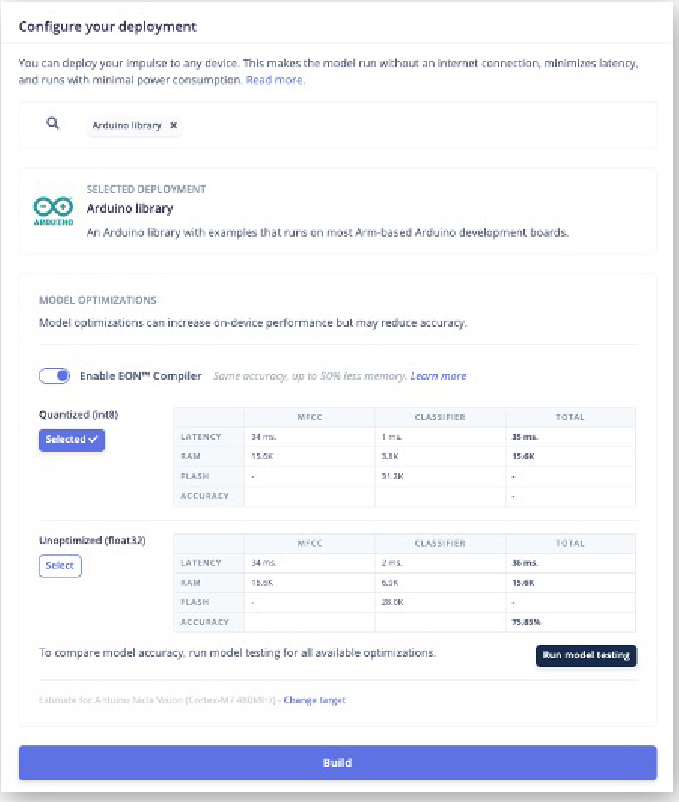
For NO, we got 0.72, which was expected, but to improve this result, we can move the samples that were not correctly classified to the training dataset and then repeat the training process.

Live Classification

We can proceed to the project's next step but also consider that it is possible to perform `Live Classification` using the NiclaV or a smartphone to capture live samples, testing the trained model before deployment on our device.

Deploy and Inference

The EIS will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. Go to the `Deployment` section, select `Arduino Library`, and at the bottom, choose `Quantized (Int8)` and press `Build`.



Configure your deployment

You can deploy your impulse to any device. This makes the model run without an internet connection, minimizes latency, and runs with minimal power consumption. Read more.

SELECTED DEPLOYMENT

Arduino library

An Arduino library with examples that runs on most Arm-based Arduino development boards.

MODEL OPTIMIZATIONS

Model optimizations can increase on-device performance but may reduce accuracy.

Enable EON™ Compiler *Same accuracy, up to 50% less memory. Learn more*

Quantized (int8) Selected ✓

	MFCC	CLASSIFIER	TOTAL
LATENCY	34 ms.	1 ms.	35 ms.
RAM	15.6K	3.8K	15.6K
FLASH	-	31.2K	-
ACCURACY			-

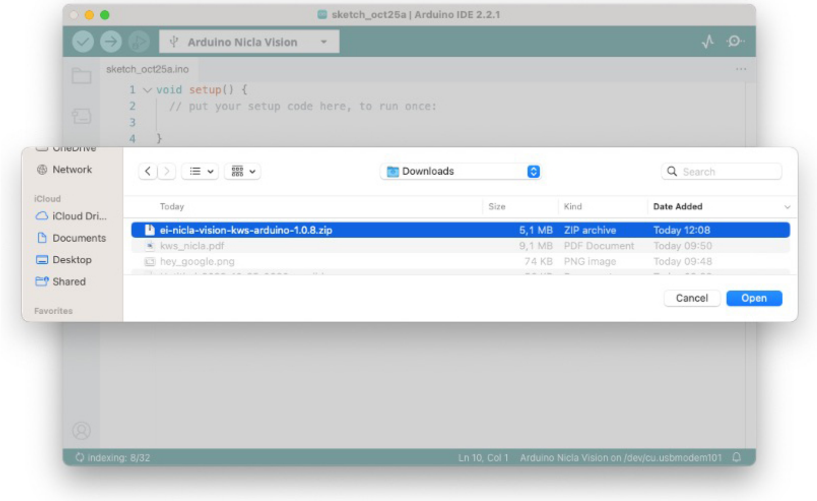
Unoptimized (float32) Select

	MFCC	CLASSIFIER	TOTAL
LATENCY	34 ms.	2 ms.	36 ms.
RAM	15.6K	6.3K	15.6K
FLASH	-	28.0K	-
ACCURACY			75.85%

To compare model accuracy, run model testing for all available optimizations.

Estimate for Arduino Nicla Vision (Cortex-M7 480MHz) - Change target

When the `Build` button is selected, a zip file will be created and downloaded to your computer. On your Arduino IDE, go to the `Sketch` tab, select the option `Add .ZIP Library`, and Choose the .zip file downloaded by EIS:

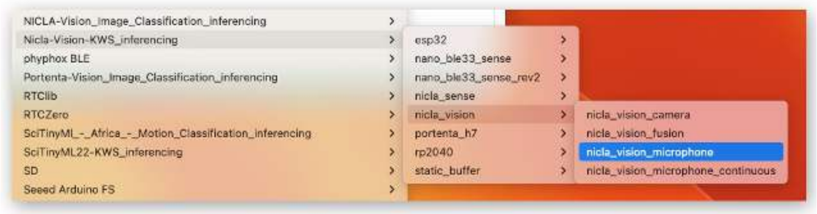


```
void setup() {
  // put your setup code here, to run once:

}
```

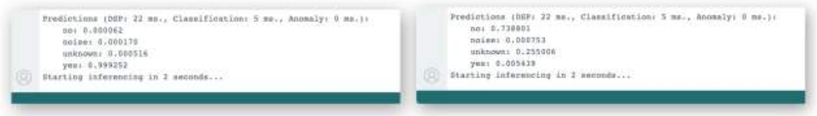
Today	Size	Kind	Date Added
ei-nicla-vision-kws-arduino-1.0.8.zip	5,1 MB	ZIP archive	Today 12:08
kws_nicla.pdf	9,1 MB	PDF Document	Today 09:50
hey_google.png	74 KB	PNG image	Today 09:48

Now, it is time for a real test. We will make inferences while completely disconnected from the EIS. Let's use the NiclaV code example created when we deployed the Arduino Library.

In your Arduino IDE, go to the `File/Examples` tab, look for your project, and select `nicla-vision/nicla-vision_microphone` (or `nicla-vision_microphone_continuous`)


NICLA-Vision_Image_Classification_inferencing
Nicla-Vision-KWS_inferencing
phyphox BLE
Portenta-Vision_Image_Classification_inferencing
RTClib
RTCZero
SciTinyML_-_Africa_-_Motion_Classification_inferencing
SciTinyML22-KWS_inferencing
SD
Seeed Arduino FS
esp32
nano_ble33_sense
nano_ble33_sense_rev2
nicla_sense
nicla_vision
portenta_h7
rp2040
static_buffer
nicla_vision_camera
nicla_vision_fusion
nicla_vision_microphone
nicla_vision_microphone_continuous

Press the reset button twice to put the NiclaV in boot mode, upload the sketch to your board, and test some real inferences:



```
Predictions (DSP: 22 ms., Classification: 5 ms., Anomaly: 0 ms.):
    no: 0.000062
    noise: 0.000179
    unknown: 0.000516
    yes: 0.999252
Starting inferencing in 2 seconds...
```

```
Predictions (DSP: 22 ms., Classification: 5 ms., Anomaly: 0 ms.):
    no: 0.738801
    noise: 0.000753
    unknown: 0.255006
    yes: 0.005439
Starting inferencing in 2 seconds...
```

Post-processing

Now that we know the model is working since it detects our keywords, let's modify the code to see the result with the NiclaV completely offline (disconnected from the PC and powered by a battery, a power bank, or an independent 5V power supply).

The idea is that whenever the keyword YES is detected, the Green LED will light; if a NO is heard, the Red LED will light, if it is a UNKNOWN, the Blue LED will light; and in the presence of noise (No Keyword), the LEDs will be OFF.

We should modify one of the code examples. Let's do it now with the `nicla-vision_microphone_continuous`.

Start with initializing the LEDs:

```
...
void setup()
{
        // Once you finish debugging your code, you can
        // comment or delete the Serial part of the code
    Serial.begin(115200);
    while (!Serial);
    Serial.println("Inferencing - Nicla Vision KWS with LEDs");

    // Pins for the built-in RGB LEDs on the Arduino NiclaV
    pinMode(LEDR, OUTPUT);
    pinMode(LEDG, OUTPUT);
    pinMode(LEDB, OUTPUT);

    // Ensure the LEDs are OFF by default.
    // Note: The RGB LEDs on the Arduino Nicla Vision
    // are ON when the pin is LOW, OFF when HIGH.
    digitalWrite(LEDR, HIGH);
    digitalWrite(LEDG, HIGH);
    digitalWrite(LEDB, HIGH);
...
}
```

Create two functions, `turn_off_leds()` function , to turn off all RGB LEDs

```
/*
 * @brief      turn_off_leds function - turn-off all RGB LEDs
 */
void turn_off_leds(){
    digitalWrite(LEDR, HIGH);
    digitalWrite(LEDG, HIGH);
    digitalWrite(LEDB, HIGH);
}
```

Another `turn_on_led()` function is used to turn on the RGB LEDs according to the most probable result of the classifier.

```
/*
 * @brief     turn_on_leds function used to turn on the RGB LEDs
 * @param[in] pred_index
 *            no:       [0] ==> Red ON
 *            noise:    [1] ==> ALL OFF
 *            unknown:  [2] ==> Blue ON
 *            Yes:      [3] ==> Green ON
 */
void turn_on_leds(int pred_index) {
  switch (pred_index)
  {
    case 0:
      turn_off_leds();
      digitalWrite(LEDR, LOW);
      break;

    case 1:
      turn_off_leds();
      break;

    case 2:
      turn_off_leds();
      digitalWrite(LEDB, LOW);
      break;

    case 3:
      turn_off_leds();
      digitalWrite(LEDG, LOW);
      break;
  }
}
```

And change the `// print the predictions` portion of the code on `loop()`:

```
...

 if (++print_results >= (EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW)) {
     // print the predictions
     ei_printf("Predictions ");
     ei_printf("(DSP: %d ms., Classification: %d ms.,
                Anomaly: %d ms.)",
         result.timing.dsp, result.timing.classification,
         result.timing.anomaly);
     ei_printf(": \n");
     int pred_index = 0;     // Initialize pred_index
     float pred_value = 0;   // Initialize pred_value
     for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
         if (result.classification[ix].value > pred_value){
             pred_index = ix;
             pred_value = result.classification[ix].value;
         }
         // ei_printf("    %s: ",
         // result.classification[ix].label);
         // ei_printf_float(result.classification[ix].value);
         // ei_printf("\n");
     }
     ei_printf("  PREDICTION: ==> %s with probability %.2f\n",
               result.classification[pred_index].label,
               pred_value);
     turn_on_leds (pred_index);


#if EI_CLASSIFIER_HAS_ANOMALY == 1
        ei_printf("    anomaly score: ");
        ei_printf_float(result.anomaly);
        ei_printf("\n");
#endif

        print_results = 0;
    }
}

...
```

You can find the complete code on the project's GitHub.

Upload the sketch to your board and test some real inferences. The idea is that the Green LED will be ON whenever the keyword YES is detected, the Red will lit for a NO, and any other word will turn on the Blue LED. All the LEDs should be off if silence or background noise is present. Remember that the same procedure can "trigger" an external device to perform a desired action instead of turning on an LED, as we saw in the introduction.

https://youtu.be/25Rd76OTXLY

Conclusion

> You will find the notebooks and codeused in this hands-on tutorial on the GitHub repository.

Before we finish, consider that Sound Classification is more than just voice. For example, you can develop TinyML projects around sound in several areas, such as:

- **Security** (Broken Glass detection, Gunshot)
- **Industry** (Anomaly Detection)
- **Medical** (Snore, Cough, Pulmonary diseases)

- **Nature** (Beehive control, insect sound, pouching mitigation)

Resources

- Subset of Google Speech Commands Dataset
- KWS MFCC Analysis Colab Notebook
- KWS_CNN_training Colab Notebook
- Arduino Post-processing Code
- Edge Impulse Project

Motion Classification and Anomaly Detection

Figure 20.7: *DALL·E 3 Prompt: 1950s style cartoon illustration depicting a movement research room. In the center of the room, there's a simulated container used for transporting goods on trucks, boats, and forklifts. The container is detailed with rivets and markings typical of industrial cargo boxes. Around the container, the room is filled with vintage equipment, including an oscilloscope, various sensor arrays, and large paper rolls of recorded data. The walls are adorned with educational posters about transportation safety and logistics. The overall ambiance of the room is nostalgic and scientific, with a hint of industrial flair.*

Overview

Transportation is the backbone of global commerce. Millions of containers are transported daily via various means, such as ships, trucks, and trains, to destinations worldwide. Ensuring these containers' safe and efficient transit is a monumental task that requires leveraging modern technology, and TinyML is undoubtedly one of them.

In this hands-on tutorial, we will work to solve real-world problems related to transportation. We will develop a Motion Classification and Anomaly Detection system using the Arduino Nicla Vision board, the Arduino IDE, and the Edge Impulse Studio. This project will help us understand how containers experience different forces and motions during various phases of transportation, such as terrestrial and maritime transit, vertical movement via forklifts, and stationary periods in warehouses.

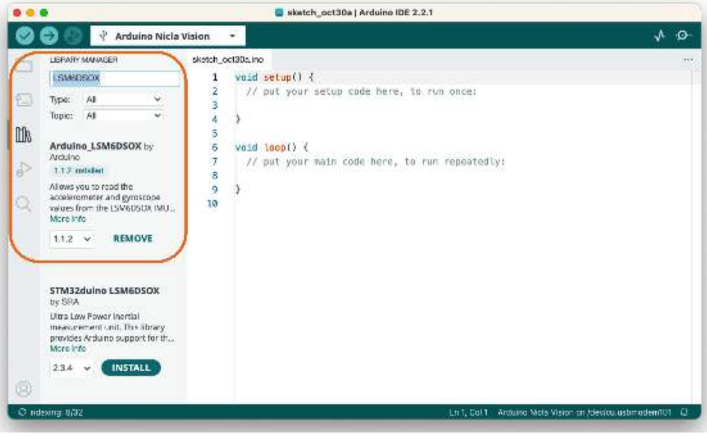
Learning Objectives

- Setting up the Arduino Nicla Vision Board
- Data Collection and Preprocessing
- Building the Motion Classification Model
- Implementing Anomaly Detection
- Real-world Testing and Analysis

By the end of this tutorial, you'll have a working prototype that can classify different types of motion and detect anomalies during the transportation of containers. This knowledge can be a stepping stone to more advanced projects in the burgeoning field of TinyML involving vibration.

IMU Installation and testing

For this project, we will use an accelerometer. As discussed in the Hands-On Tutorial, *Setup Nicla Vision*, the Nicla Vision Board has an onboard **6-axis IMU**: 3D gyroscope and 3D accelerometer, the LSM6DSOX. Let's verify if the LSM6DSOX IMU library is installed. If not, install it.



LSM6DSOX

Type: All

Topic: All

Arduino_LSM6DSOX by Arduino

1.1.2 installed

Allows you to read the accelerometer and gyroscope values from the LSM6DSOX IMU...
More info

1.1.2 REMOVE

STM32duino LSM6DSOX by SRA

Ultra Low Power inertial measurement unit. This library provides Arduino support for th...
More info

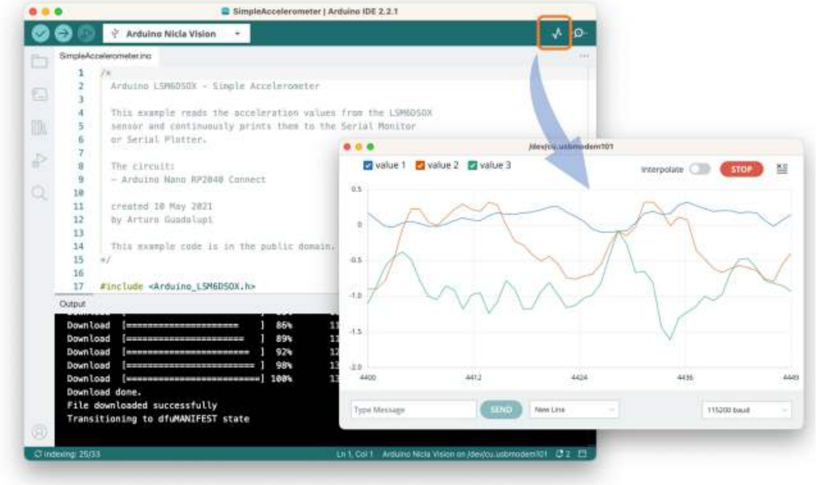
2.3.4 INSTALL

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Indexing: 8/32

Ln 1, Col 1   Arduino Nicla Vision on /dev/cu.usbmodem101

Next, go to `Examples > Arduino_LSM6DSOX > SimpleAccelerometer` and run the accelerometer test. You can check if it works by opening the IDE Serial Monitor or Plotter. The values are in g (earth gravity), with a default range of +/- 4g:



```
/*
  Arduino LSM6DSOX - Simple Accelerometer

  This example reads the acceleration values from the LSM6DSOX
  sensor and continuously prints them to the Serial Monitor
  or Serial Plotter.

  The circuit:
  - Arduino Nano RP2040 Connect

  created 10 May 2021
  by Arturo Guadalupi

  This example code is in the public domain.
*/

#include <Arduino_LSM6DSOX.h>
```

```
Download  [=====================      ]  86%
Download  [======================     ]  89%
Download  [=======================    ]  92%
Download  [========================   ]  98%
Download  [===========================] 100%
Download done.
File downloaded successfully
Transitioning to dfuMANIFEST state
```

Defining the Sampling frequency:

Choosing an appropriate sampling frequency is crucial for capturing the motion characteristics you're interested in studying. The Nyquist-Shannon sampling theorem states that the sampling rate should be at least twice the highest frequency component in the signal to reconstruct it properly. In the context of motion classification and anomaly detection for transportation, the choice of sampling frequency would depend on several factors:

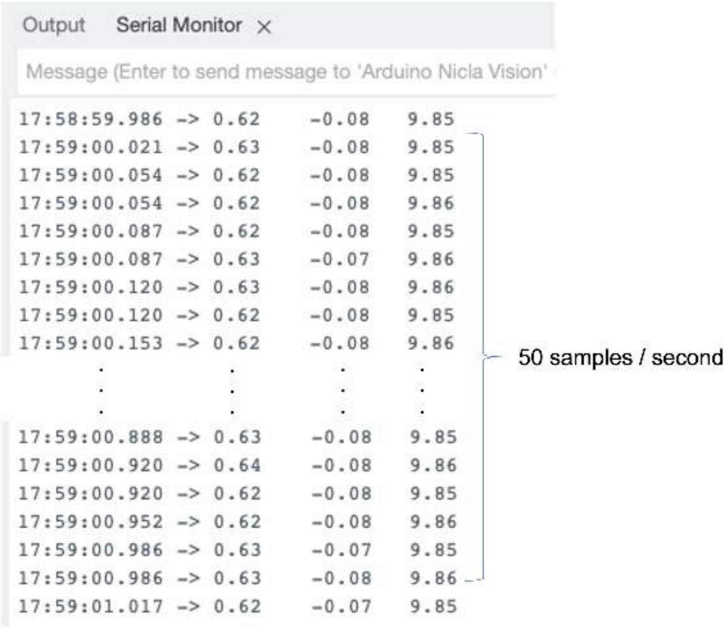
1. **Nature of the Motion**: Different types of transportation (terrestrial, maritime, etc.) may involve different ranges of motion frequencies. Faster movements may require higher sampling frequencies.
2. **Hardware Limitations**: The Arduino Nicla Vision board and any associated sensors may have limitations on how fast they can sample data.
3. **Computational Resources**: Higher sampling rates will generate more data, which might be computationally intensive, especially critical in a TinyML environment.
4. **Battery Life**: A higher sampling rate will consume more power. If the system is battery-operated, this is an important consideration.
5. **Data Storage**: More frequent sampling will require more storage space, another crucial consideration for embedded systems with limited memory.

In many human activity recognition tasks, **sampling rates of around 50 Hz to 100 Hz** are commonly used. Given that we are simulating transportation scenarios, which are generally not high-frequency events, a sampling rate in that range (50-100 Hz) might be a reasonable starting point.

Let's define a sketch that will allow us to capture our data with a defined sampling frequency (for example, 50 Hz):

```
/*
 * Based on Edge Impulse Data Forwarder Example (Arduino)
  - https://docs.edgeimpulse.com/docs/cli-data-forwarder
 * Developed by M.Rovai @11May23
 */

/* Include ------------------------------------------- */
#include <Arduino_LSM6DSOX.h>

/* Constant defines ---------------------------------- */
#define CONVERT_G_TO_MS2 9.80665f
#define FREQUENCY_HZ        50
#define INTERVAL_MS         (1000 / (FREQUENCY_HZ + 1))

static unsigned long last_interval_ms = 0;
float x, y, z;

void setup() {
  Serial.begin(9600);
  while (!Serial);

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }
}

void loop() {
  if (millis() > last_interval_ms + INTERVAL_MS) {
    last_interval_ms = millis();

    if (IMU.accelerationAvailable()) {
      // Read raw acceleration measurements from the device
      IMU.readAcceleration(x, y, z);

      // converting to m/s2
      float ax_m_s2 = x * CONVERT_G_TO_MS2;
      float ay_m_s2 = y * CONVERT_G_TO_MS2;
      float az_m_s2 = z * CONVERT_G_TO_MS2;

      Serial.print(ax_m_s2);
      Serial.print("\t");
      Serial.print(ay_m_s2);
      Serial.print("\t");
      Serial.println(az_m_s2);
    }
  }
}
```

Uploading the sketch and inspecting the Serial Monitor, we can see that we are capturing 50 samples per second.



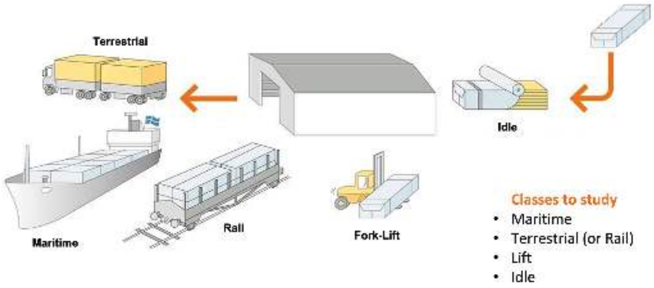
```
17:58:59.986 -> 0.62    -0.08    9.85
17:59:00.021 -> 0.63    -0.08    9.85
17:59:00.054 -> 0.62    -0.08    9.85
17:59:00.054 -> 0.62    -0.08    9.86
17:59:00.087 -> 0.62    -0.08    9.85
17:59:00.087 -> 0.63    -0.07    9.86
17:59:00.120 -> 0.63    -0.08    9.86
17:59:00.120 -> 0.62    -0.08    9.85
17:59:00.153 -> 0.62    -0.08    9.86
      .             .        .       .
      .             .        .       .
      .             .        .       .
17:59:00.888 -> 0.63    -0.08    9.85
17:59:00.920 -> 0.64    -0.08    9.86
17:59:00.920 -> 0.62    -0.08    9.85
17:59:00.952 -> 0.62    -0.08    9.86
17:59:00.986 -> 0.63    -0.07    9.85
17:59:00.986 -> 0.63    -0.08    9.86
17:59:01.017 -> 0.62    -0.07    9.85
```

Note that with the Nicla board resting on a table (with the camera facing down), the z-axis measures around 9.8 $\mathrm{m/s^2}$, the expected earth acceleration.

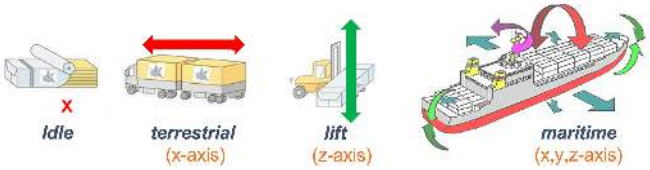
The Case Study: Simulated Container Transportation

We will simulate container (or better package) transportation through different scenarios to make this tutorial more relatable and practical. Using the built-in accelerometer of the Arduino Nicla Vision board, we'll capture motion data by manually simulating the conditions of:

1. **Terrestrial** Transportation (by road or train)
2. **Maritime**-associated Transportation
3. Vertical Movement via Fork-**Lift**
4. Stationary **(Idle)** period in a Warehouse



From the above images, we can define for our simulation that primarily horizontal movements (x or y axis) should be associated with the "Terrestrial class," Vertical movements (z-axis) with the "Lift Class," no activity with the "Idle class," and movement on all three axes to Maritime class.



Data Collection

For data collection, we can have several options. In a real case, we can have our device, for example, connected directly to one container, and the data collected on a file (for example .CSV) and stored on an SD card (Via SPI connection) or an offline repo in your computer. Data can also be sent remotely to a nearby repository, such as a mobile phone, using Bluetooth (as done in this project: Sensor DataLogger). Once your dataset is collected and stored as a .CSV file, it can be uploaded to the Studio using the CSV Wizard tool.

> In this video, you can learn alternative ways to send data to the Edge Impulse Studio.

Connecting the device to Edge Impulse

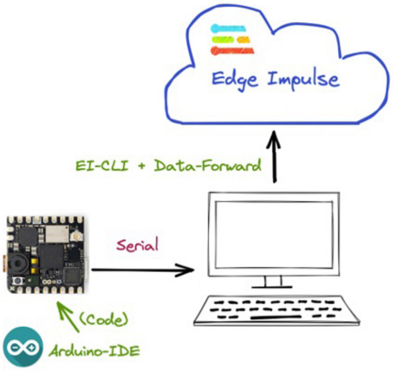
We will connect the Nicla directly to the Edge Impulse Studio, which will also be used for data pre-processing, model training, testing, and deployment. For that, you have two options:

1. Download the latest firmware and connect it directly to the `Data Collection` section.
2. Use the CLI Data Forwarder tool to capture sensor data from the sensor and send it to the Studio.

Option 1 is more straightforward, as we saw in the *Setup Nicla Vision* hands-on, but option 2 will give you more flexibility regarding capturing your data, such as sampling frequency definition. Let's do it with the last one.

Please create a new project on the Edge Impulse Studio (EIS) and connect the Nicla to it, following these steps:

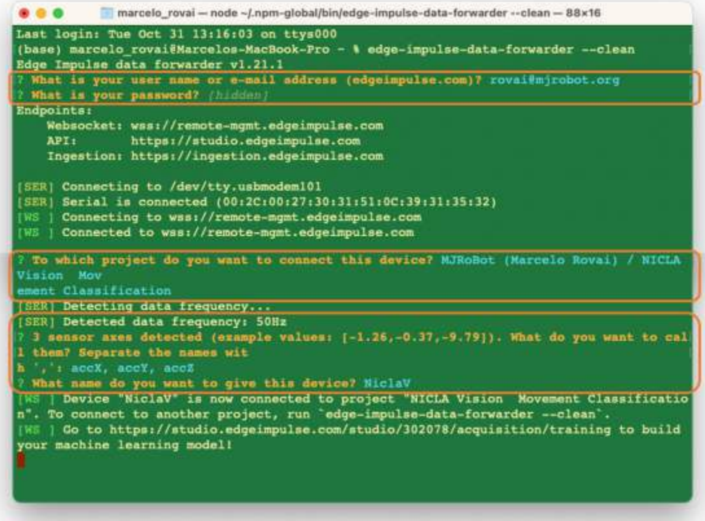
1. Install the Edge Impulse CLI and the Node.js into your computer.
2. Upload a sketch for data capture (the one discussed previously in this tutorial).
3. Use the CLI Data Forwarder to capture data from the Nicla's accelerometer and send it to the Studio, as shown in this diagram:



Start the CLI Data Forwarder on your terminal, entering (if it is the first time) the following command:

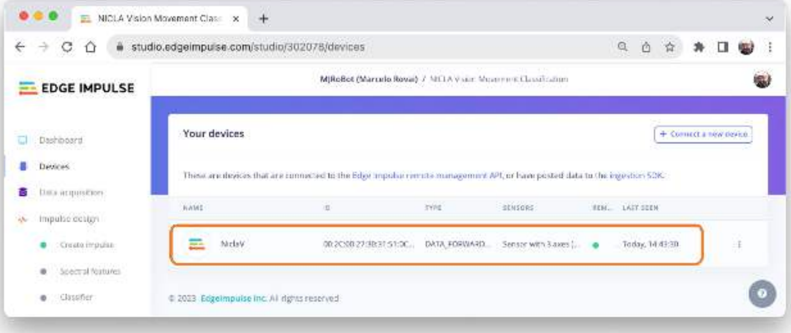
```
$ edge-impulse-data-forwarder --clean
```

Next, enter your EI credentials and choose your project, variables (for example, *accX*, *accY*, and *accZ*), and device name (for example, *NiclaV*:


marcelo_rovai — node ~/.npm-global/bin/edge-impulse-data-forwarder --clean — 88×16

```
Last login: Tue Oct 31 13:16:03 on ttys000
(base) marcelo_rovai@Marcelos-MacBook-Pro ~ % edge-impulse-data-forwarder --clean
Edge Impulse data forwarder v1.21.1
? What is your user name or e-mail address (edgeimpulse.com)? rovai@mjrobot.org
? What is your password? [hidden]
Endpoints:
    Websocket: wss://remote-mgmt.edgeimpulse.com
    API:       https://studio.edgeimpulse.com
    Ingestion: https://ingestion.edgeimpulse.com

[SER] Connecting to /dev/tty.usbmodem101
[SER] Serial is connected (00:2C:00:27:30:31:51:0C:39:31:35:32)
[WS ] Connecting to wss://remote-mgmt.edgeimpulse.com
[WS ] Connected to wss://remote-mgmt.edgeimpulse.com

? To which project do you want to connect this device? MJRoBot (Marcelo Rovai) / NICLA
Vision  Mov
ement Classification
[SER] Detecting data frequency...
[SER] Detected data frequency: 50Hz
? 3 sensor axes detected (example values: [-1.26,-0.37,-9.79]). What do you want to cal
l them? Separate the names wit
h ',': accX, accY, accZ
? What name do you want to give this device? NiclaV
[WS ] Device "NiclaV" is now connected to project "NICLA Vision  Movement Classificatio
n". To connect to another project, run `edge-impulse-data-forwarder --clean`.
[WS ] Go to https://studio.edgeimpulse.com/studio/302078/acquisition/training to build
your machine learning model!
```

Go to the `Devices` section on your EI Project and verify if the device is connected (the dot should be green):



studio.edgeimpulse.com/studio/302078/devices

MjRoBot (Marcelo Rovai) / NICLA Vision Movement Classification

Your devices

These are devices that are connected to the Edge Impulse remote management API, or have posted data to the ingestion SDK.

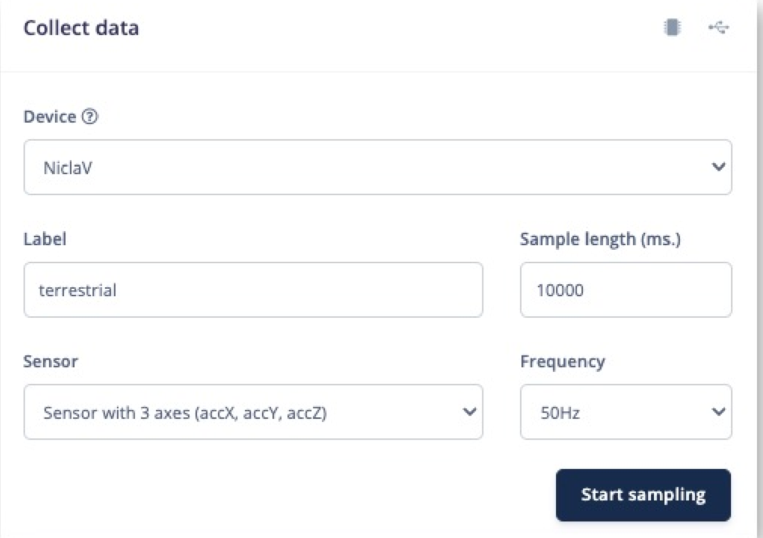
NAME	ID	TYPE	SENSORS	REM...	LAST SEEN
NiclaV	00:2C:00:27:38:31:51:0C...	DATA_FORWARD...	Sensor with 3 axes (...	●	Today, 14:43:30

© 2023 EdgeImpulse Inc. All rights reserved

> You can clone the project developed for this hands-on: NICLA Vision Movement Classification.

Data Collection

On the `Data Acquisition` section, you should see that your board `[NiclaV]` is connected. The sensor is available: `[sensor with 3 axes (accX, accY, accZ)]` with a sampling frequency of `[50 Hz]`. The Studio suggests a sample length of `[10000]` ms (10 s). The last thing left is defining the sample label. Let's start with `[terrestrial]`:



Collect data

Device: NiclaV

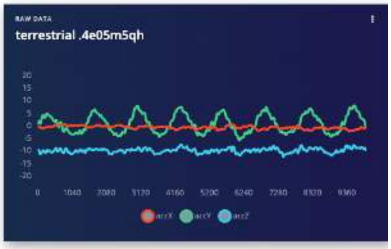
Label: terrestrial

Sample length (ms.): 10000

Sensor: Sensor with 3 axes (accX, accY, accZ)

Frequency: 50Hz

Terrestrial (palettes in a Truck or Train), moving horizontally. Press `[Start Sample]` and move your device horizontally, keeping one direction over your table. After 10 s, your data will be uploaded to the studio. Here is how the sample was collected:

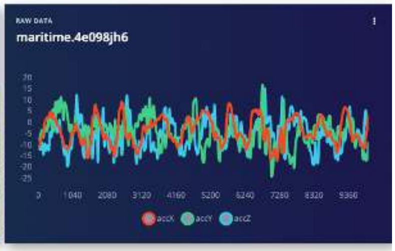


As expected, the movement was captured mainly in the Y-axis (green). In the blue, we see the Z axis, around -10 m/s^2 (the Nicla has the camera facing up).

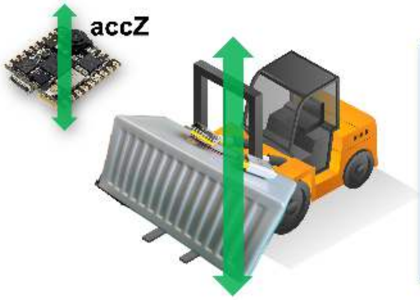
As discussed before, we should capture data from all four Transportation Classes. So, imagine that you have a container with a built-in accelerometer facing the following situations:

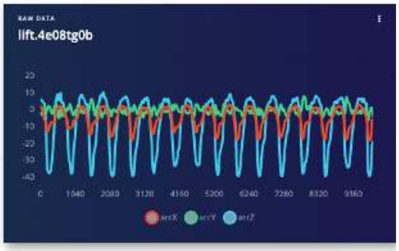
Maritime (pallets in boats into an angry ocean). The movement is captured on all three axes:

accX, accY and accZ

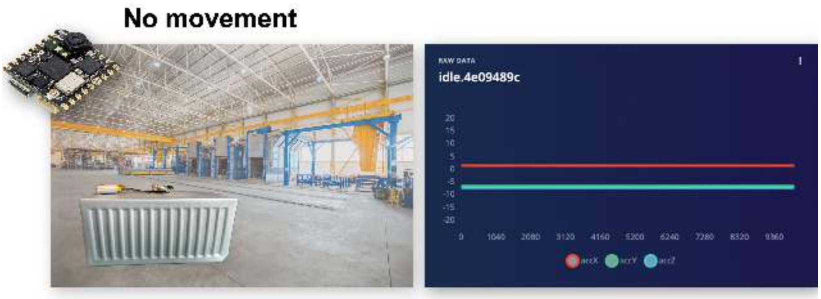


Lift (Palettes being handled vertically by a Forklift). Movement captured only in the Z-axis:

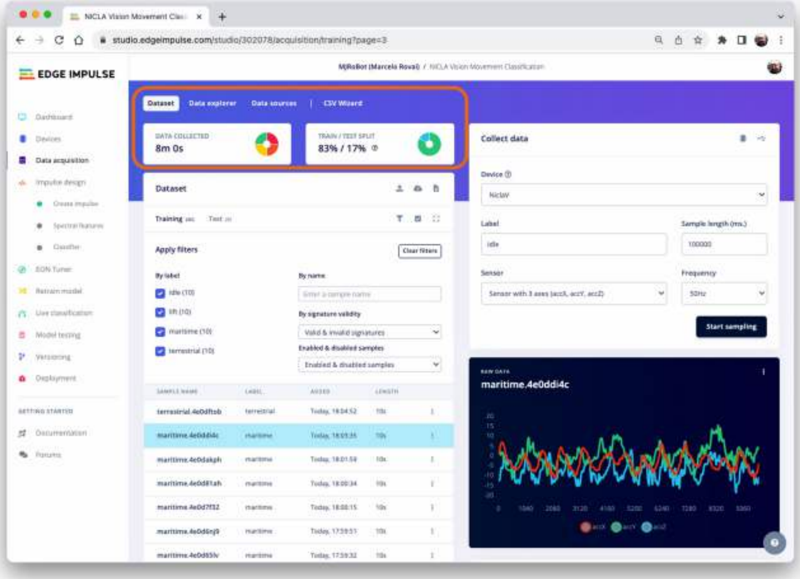




Idle (Paletts in a warehouse). No movement detected by the accelerometer:



You can capture, for example, 2 minutes (twelve samples of 10 seconds) for each of the four classes (a total of 8 minutes of data). Using the `three dots` menu after each one of the samples, select 2 of them, reserving them for the Test set. Alternatively, you can use the automatic `Train/Test Split tool` on the `Danger Zone` of `Dashboard` tab. Below, you can see the resulting dataset:



DATA COLLECTED
8m 0s

TRAIN / TEST SPLIT
83% / 17%

Dataset

Training Test

Apply filters

By label

☑ idle (10)
☑ lift (10)
☑ maritime (10)
☑ terrestrial (10)

By name: Enter a sample name

By signature validity: Valid & invalid signatures

Enabled & disabled samples: Enabled & disabled samples

SAMPLE NAME	LABEL	ADDED	LENGTH
terrestrial.4e0dftob	terrestrial	Today, 18:04:52	10s
maritime.4e0ddi4c	maritime	Today, 18:03:35	10s
maritime.4e0dakph	maritime	Today, 18:01:59	10s
maritime.4e0d81ah	maritime	Today, 18:00:34	10s
maritime.4e0d7f32	maritime	Today, 18:00:15	10s
maritime.4e0d6nj9	maritime	Today, 17:59:51	10s
maritime.4e0d65lv	maritime	Today, 17:59:32	10s

Collect data

Device: NiclaV

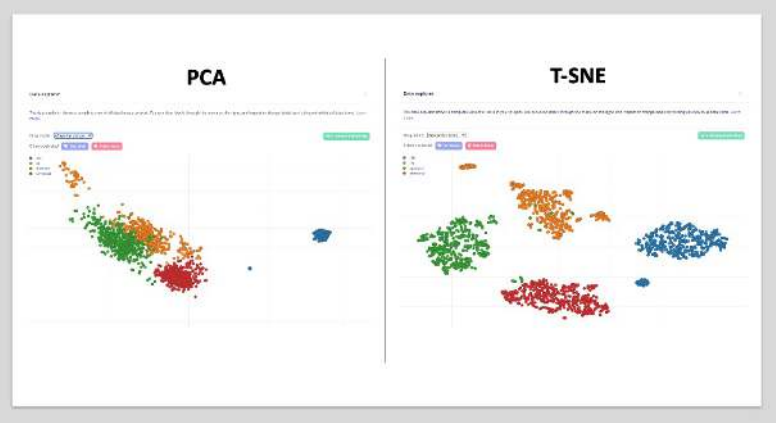
Label: idle

Sample length (ms.): 100000

Sensor: Sensor with 3 axes (accX, accY, accZ)

Frequency: 50Hz

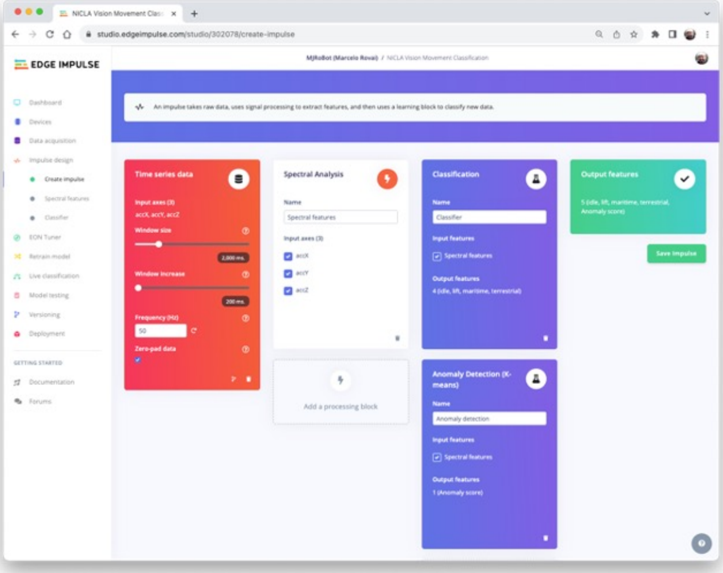
Once you have captured your dataset, you can explore it in more detail using the Data Explorer, a visual tool to find outliers or mislabeled data (helping to correct them). The data explorer first tries to extract meaningful features from your data (by applying signal processing and neural network embeddings) and then uses a dimensionality reduction algorithm such as PCA or t-SNE to map these features to a 2D space. This gives you a one-look overview of your complete dataset.



In our case, the dataset seems OK (good separation). But the PCA shows we can have issues between maritime (green) and lift (orange). This is expected, once on a boat, sometimes the movement can be only "vertical".

Impulse Design

The next step is the definition of our Impulse, which takes the raw data and uses signal processing to extract features, passing them as the input tensor of a *learning block* to classify new data. Go to `Impulse Design` and `Create Impulse`. The Studio will suggest the basic design. Let's also add a second *Learning Block* for `Anomaly Detection`.



An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

Time series data
Input axes (3)
accX, accY, accZ
Window size: 2,000 ms.
Window increase: 200 ms.
Frequency (Hz): 50
Zero-pad data: ☑

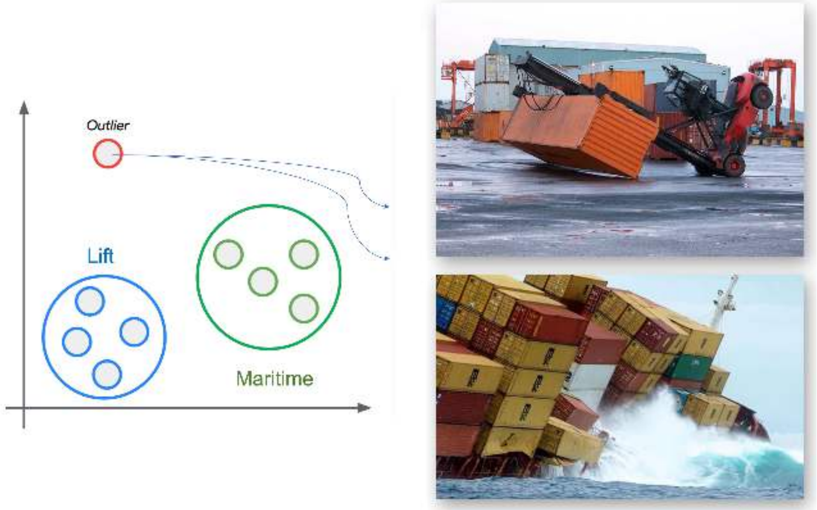
Spectral Analysis
Name: Spectral features
Input axes (3)
☑ accX
☑ accY
☑ accZ

Classification
Name: Classifier
Input features
☑ Spectral features
Output features
4 (idle, lift, maritime, terrestrial)

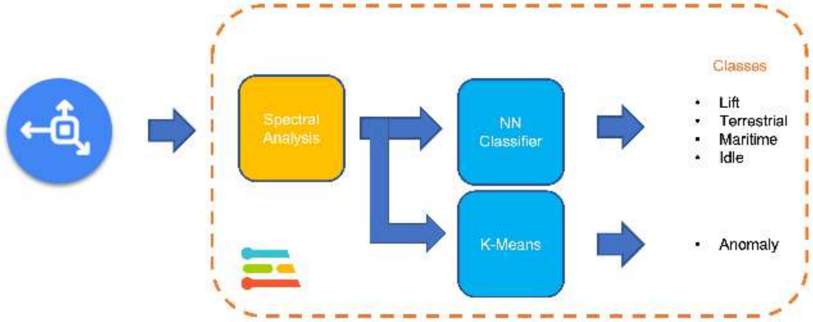
Anomaly Detection (K-means)
Name: Anomaly detection
Input features
☑ Spectral features
Output features
1 (Anomaly score)

Output features
5 (idle, lift, maritime, terrestrial, Anomaly score)

This second model uses a K-means model. If we imagine that we could have our known classes as clusters, any sample that could not fit on that could be an outlier, an anomaly such as a container rolling out of a ship on the ocean or falling from a Forklift.



The sampling frequency should be automatically captured, if not, enter it: [50] Hz. The Studio suggests a *Window Size* of 2 seconds ([2000] ms) with a *sliding window* of [20] ms. What we are defining in this step is that we will pre-process the captured data (Time-Seres data), creating a tabular dataset features) that will be the input for a Neural Networks Classifier (DNN) and an Anomaly Detection model (K-Means), as shown below:



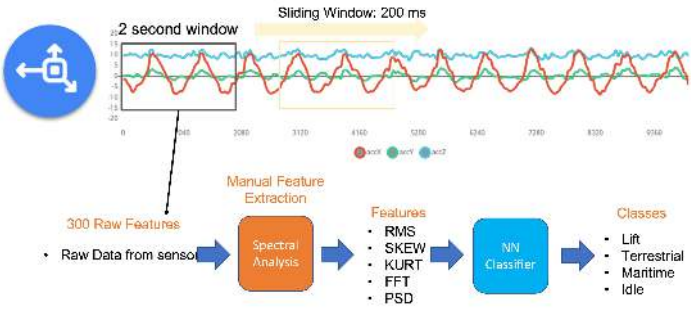
Let's dig into those steps and parameters to understand better what we are doing here.

Data Pre-Processing Overview

Data pre-processing is extracting features from the dataset captured with the accelerometer, which involves processing and analyzing the raw data. Accelerometers measure the acceleration of an object along one or more axes (typically three, denoted as X, Y, and Z). These measurements can be used to understand various aspects of the object's motion, such as movement patterns and vibrations.

Raw accelerometer data can be noisy and contain errors or irrelevant information. Preprocessing steps, such as filtering and normalization, can clean and standardize the data, making it more suitable for feature extraction. In our case, we should divide the data into smaller segments or **windows**. This can help focus on specific events or activities within the dataset, making feature extraction more manageable and meaningful. The **window size** and overlap (**window increase**) choice depend on the application and the frequency of the events of interest. As a thumb rule, we should try to capture a couple of "cycles of data".

> With a sampling rate (SR) of 50 Hz and a window size of 2 seconds, we will get 100 samples per axis, or 300 in total (3 axis × 2 seconds × 50 samples). We will slide this window every 200 ms, creating a larger dataset where each instance has 300 raw features.



Once the data is preprocessed and segmented, you can extract features that describe the motion's characteristics. Some typical features extracted from accelerometer data include:

- **Time-domain** features describe the data's statistical properties within each segment, such as mean, median, standard deviation, skewness, kurtosis, and zero-crossing rate.
- **Frequency-domain** features are obtained by transforming the data into the frequency domain using techniques like the Fast Fourier Transform (FFT). Some typical frequency-domain features include the power spectrum, spectral energy, dominant frequencies (amplitude and frequency), and spectral entropy.
- **Time-frequency** domain features combine the time and frequency domain information, such as the Short-Time Fourier Transform (STFT) or the Discrete Wavelet Transform (DWT). They can provide a more detailed understanding of how the signal's frequency content changes over time.

In many cases, the number of extracted features can be large, which may lead to overfitting or increased computational complexity. Feature selection techniques, such as mutual information, correlation-based methods, or principal component analysis (PCA), can help identify the most relevant features for a given application and reduce the dimensionality of the dataset. The Studio can help with such feature importance calculations.

EI Studio Spectral Features

Data preprocessing is a challenging area for embedded machine learning, still, Edge Impulse helps overcome this with its digital signal processing (DSP) preprocessing step and, more specifically, the Spectral Features Block.

On the Studio, the collected raw dataset will be the input of a Spectral Analysis block, which is excellent for analyzing repetitive motion, such as data from accelerometers. This block will perform a DSP (Digital Signal Processing), extracting features such as FFT or Wavelets.

For our project, once the time signal is continuous, we should use FFT with, for example, a length of `[32]`.

The per axis/channel **Time Domain Statistical features** are:

- RMS: 1 feature
- Skewness: 1 feature
- Kurtosis: 1 feature

The per axis/channel **Frequency Domain Spectral features** are:

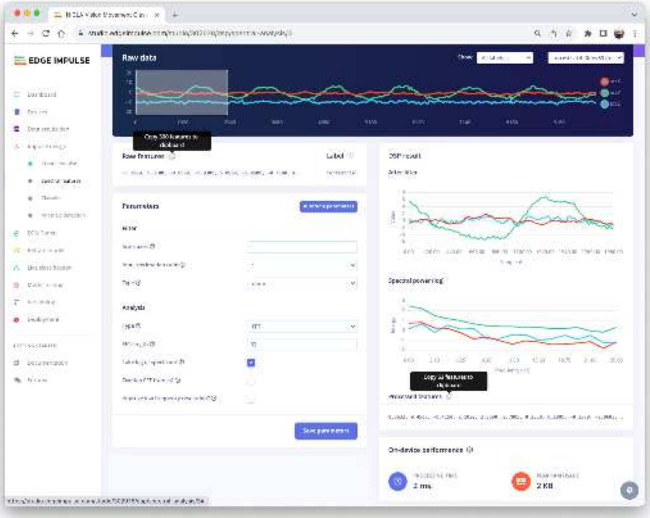
- Spectral Power: 16 features (FFT Length/2)
- Skewness: 1 feature
- Kurtosis: 1 feature

So, for an FFT length of 32 points, the resulting output of the Spectral Analysis Block will be 21 features per axis (a total of 63 features).

> You can learn more about how each feature is calculated by downloading the notebook Edge Impulse - Spectral Features Block Analysis TinyML under the hood: Spectral Analysis or opening it directly on Google CoLab.

Generating features

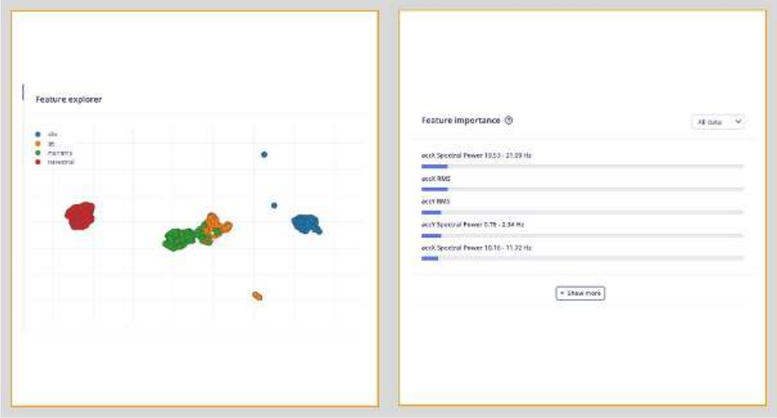
Once we understand what the pre-processing does, it is time to finish the job. So, let's take the raw data (time-series type) and convert it to tabular data. For that, go to the `Spectral Features` section on the `Parameters` tab, define the main parameters as discussed in the previous section (`[FFT]` with `[32]` points), and select`[Save Parameters]`:



At the top menu, select the `Generate Features` option and the `Generate Features` button. Each 2-second window data will be converted into one data point of 63 features.

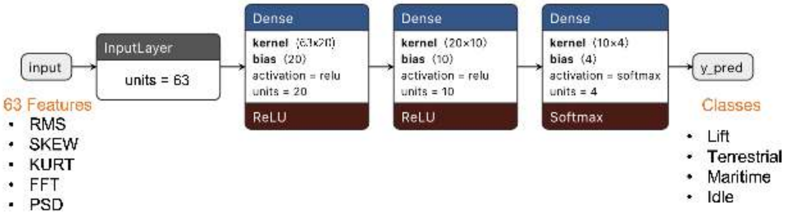
> The Feature Explorer will show those data in 2D using UMAP. Uniform Manifold Approximation and Projection (UMAP) is a dimension reduction technique that can be used for visualization similarly to t-SNE but is also applicable for general non-linear dimension reduction.

The visualization makes it possible to verify that after the feature generation, the classes present keep their excellent separation, which indicates that the classifier should work well. Optionally, you can analyze how important each one of the features is for one class compared with others.

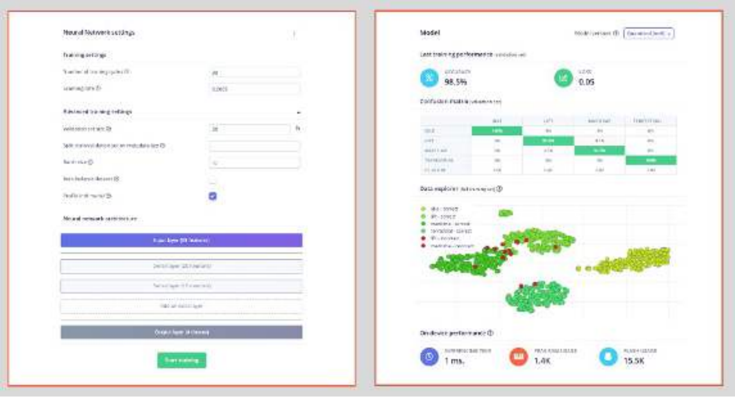


Models Training

Our classifier will be a Dense Neural Network (DNN) that will have 63 neurons on its input layer, two hidden layers with 20 and 10 neurons, and an output layer with four neurons (one per each class), as shown here:



As hyperparameters, we will use a Learning Rate of [0.005], a Batch size of [32], and [20]% of data for validation for [30] epochs. After training, we can see that the accuracy is 98.5%. The cost of memory and latency is meager.


Neural Network settings

Model

98.5%

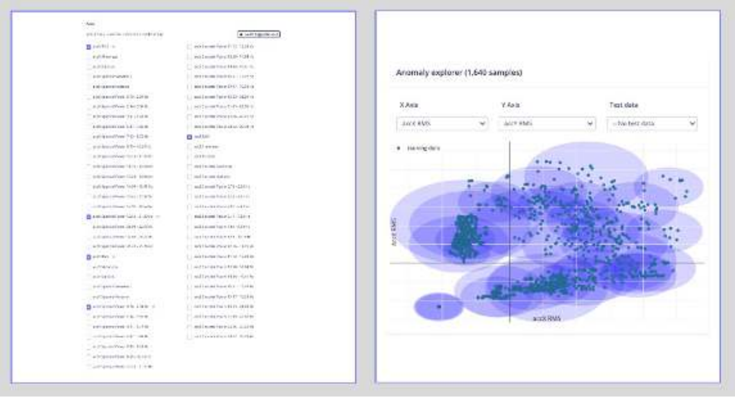
0.06

1 ms.

1.4K

15.5K

For Anomaly Detection, we will choose the suggested features that are precisely the most important ones in the Feature Extraction, plus the accZ RMS. The number of clusters will be [32], as suggested by the Studio:


Anomaly explorer (1,640 samples)

X Axis

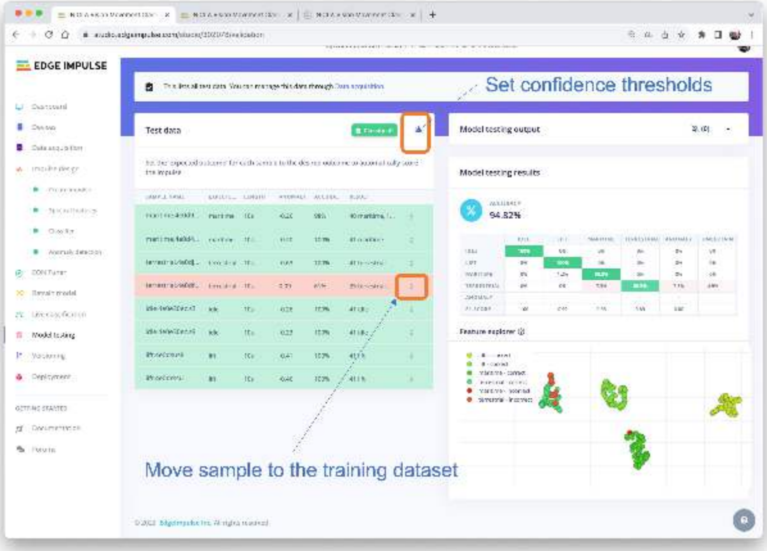
Y Axis

Test data

Testing

We can verify how our model will behave with unknown data using 20% of the data left behind during the data capture phase. The result was almost 95%, which is good. You can always work to improve the results, for example, to understand what went wrong with one of the wrong results. If it is a unique situation, you can add it to the training dataset and then repeat it.

The default minimum threshold for a considered uncertain result is `[0.6]` for classification and `[0.3]` for anomaly. Once we have four classes (their output sum should be 1.0), you can also set up a lower threshold for a class to be considered valid (for example, 0.4). You can `Set confidence thresholds` on the `three dots` menu, besides the `Classify all` button.

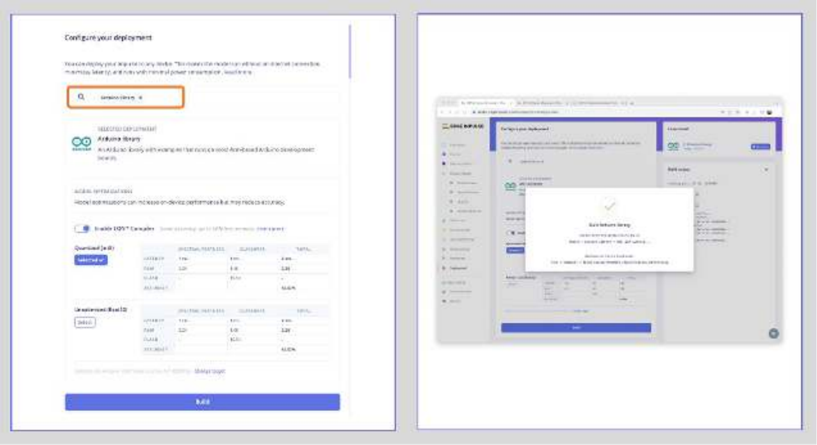


You can also perform Live Classification with your device (which should still be connected to the Studio).

> Be aware that here, you will capture real data with your device and upload it to the Studio, where an inference will be taken using the trained model (But the **model is NOT in your device**).

Deploy

It is time to deploy the preprocessing block and the trained model to the Nicla. The Studio will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. You should select the option `Arduino Library`, and at the bottom, you can choose `Quantized (Int8)` or `Unoptimized (float32)` and `[Build]`. A Zip file will be created and downloaded to your computer.

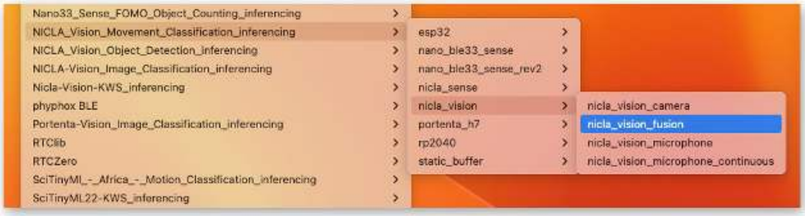


On your Arduino IDE, go to the `Sketch` tab, select `Add.ZIP Library`, and Choose the.zip file downloaded by the Studio. A message will appear in the IDE Terminal: `Library installed`.

Inference

Now, it is time for a real test. We will make inferences wholly disconnected from the Studio. Let's change one of the code examples created when you deploy the Arduino Library.

In your Arduino IDE, go to the `File/Examples` tab and look for your project, and on examples, select `Nicla_vision_fusion`:


Nano33_Sense_FOMO_Object_Counting_inferencing
NICLA_Vision_Movement_Classification_inferencing
NICLA_Vision_Object_Detection_inferencing
NICLA-Vision_Image_Classification_inferencing
Nicla-Vision-KWS_inferencing
phyphox BLE
Portenta-Vision_Image_Classification_inferencing
RTClib
RTCZero
SciTinyML_-_Africa_-_Motion_Classification_inferencing
SciTinyML22-KWS_inferencing
esp32
nano_ble33_sense
nano_ble33_sense_rev2
nicla_sense
nicla_vision
portenta_h7
rp2040
static_buffer
nicla_vision_camera
nicla_vision_fusion
nicla_vision_microphone
nicla_vision_microphone_continuous

Note that the code created by Edge Impulse considers a *sensor fusion* approach where the IMU (Accelerometer and Gyroscope) and the ToF are used. At the beginning of the code, you have the libraries related to our project, IMU and ToF:

```
/* Includes ---------------------------------------------- */
#include <NICLA_Vision_Movement_Classification_inferencing.h>
#include <Arduino_LSM6DSOX.h> //IMU
#include "VL53L1X.h" // ToF
```

> You can keep the code this way for testing because the trained model will use only features pre-processed from the accelerometer. But

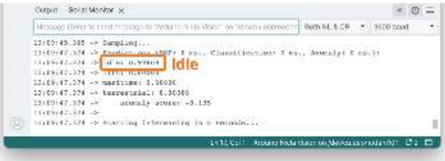
consider that you will write your code only with the needed libraries for a real project.

And that is it!

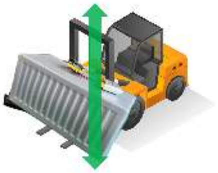
You can now upload the code to your device and proceed with the inferences. Press the Nicla `[RESET]` button twice to put it on boot mode (disconnect from the Studio if it is still connected), and upload the sketch to your board.

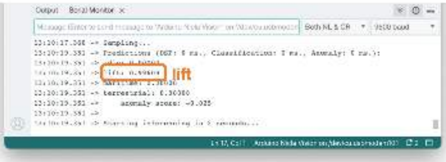
Now you should try different movements with your board (similar to those done during data capture), observing the inference result of each class on the Serial Monitor:

- **Idle and lift classes**:



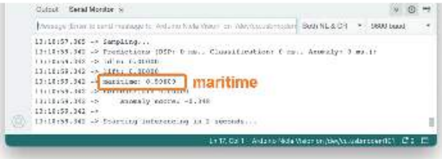
```
Output   Serial Monitor
Both NL & CR   9600 baud
13:09:45.385 -> Sampling...
13:09:47.374 -> Predictions (DSP: 6 ms., Classification: 0 ms., Anomaly: 0 ms.):
13:09:47.374 -> idle: 0.99609
13:09:47.374 -> lift: 0.00391
13:09:47.374 -> maritime: 0.00000
13:09:47.374 -> terrestrial: 0.00000
13:09:47.374 ->     anomaly score: -0.135
13:09:47.374 ->
13:09:47.374 -> Starting inferencing in 2 seconds...
```



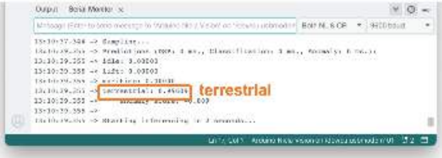


```
Output   Serial Monitor
Both NL & CR   9600 baud
13:10:17.368 -> Sampling...
13:10:19.381 -> Predictions (DSP: 6 ms., Classification: 0 ms., Anomaly: 0 ms.):
13:10:19.381 -> idle: 0.00391
13:10:19.381 -> lift: 0.99609
13:10:19.381 -> maritime: 0.00000
13:10:19.381 -> terrestrial: 0.00000
13:10:19.381 ->     anomaly score: -0.025
13:10:19.381 ->
13:10:19.381 -> Starting inferencing in 2 seconds...
```

- **Maritime and terrestrial**:



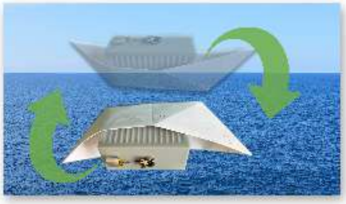
```
Output   Serial Monitor
Both NL & CR   9600 baud
13:18:57.365 -> Sampling...
13:18:59.342 -> Predictions (DSP: 0 ms., Classification: 0 ms., Anomaly: 0 ms.):
13:18:59.342 -> idle: 0.00000
13:18:59.342 -> lift: 0.00000
13:18:59.342 -> maritime: 0.99609
13:18:59.342 -> terrestrial: 0.00391
13:18:59.342 ->     anomaly score: -0.348
13:18:59.342 ->
13:18:59.342 -> Starting inferencing in 2 seconds...
```

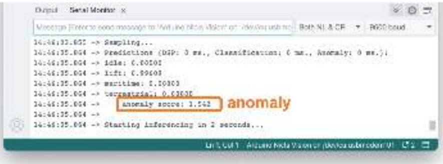


```
Output   Serial Monitor
Both NL & CR   9600 baud
13:10:27.348 -> Sampling...
13:10:29.355 -> Predictions (DSP: 0 ms., Classification: 0 ms., Anomaly: 0 ms.):
13:10:29.355 -> idle: 0.00000
13:10:29.355 -> lift: 0.00000
13:10:29.355 -> maritime: 0.00391
13:10:29.355 -> terrestrial: 0.99609
13:10:29.355 ->     anomaly score: -0.009
13:10:29.355 ->
13:10:29.355 -> Starting inferencing in 2 seconds...
```

Note that in all situations above, the value of the `anomaly score` was smaller than 0.0. Try a new movement that was not part of the original dataset, for example, "rolling" the Nicla, facing the camera upside-down, as a container falling from a boat or even a boat accident:

- **Anomaly detection**:





Output
Serial Monitor
Message (Enter to send message to 'Arduino Nicla Vision' on '/dev/cu.usbmodem...')
Both NL & CR
9600 baud

```
14:46:33.853 -> Sampling...
14:46:35.864 -> Predictions (DSP: 0 ms., Classification: 0 ms., Anomaly: 0 ms.):
14:46:35.864 -> idle: 0.00000
14:46:35.864 -> lift: 0.99609
14:46:35.864 -> maritime: 0.00000
14:46:35.864 -> terrestrial: 0.00000
14:46:35.864 ->     anomaly score: 1.542
14:46:35.864 ->
14:46:35.864 -> Starting inferencing in 2 seconds...
```

anomaly
Ln 1, Col 1   Arduino Nicla Vision on /dev/cu.usbmodem01

In this case, the anomaly is much bigger, over 1.00

Post-processing

Now that we know the model is working since it detects the movements, we suggest that you modify the code to see the result with the NiclaV completely offline (disconnected from the PC and powered by a battery, a power bank, or an independent 5 V power supply).

The idea is to do the same as with the KWS project: if one specific movement is detected, a specific LED could be lit. For example, if *terrestrial* is detected, the Green LED will light; if *maritime*, the Red LED will light, if it is a *lift,* the Blue LED will light; and if no movement is detected *(idle*), the LEDs will be OFF. You can also add a condition when an anomaly is detected, in this case, for example, a white color can be used (all e LEDs light simultaneously).

Conclusion

> The notebooks and codeused in this hands-on tutorial will be found on the GitHub repository.

Before we finish, consider that Movement Classification and Object Detection can be utilized in many applications across various domains. Here are some of the potential applications:

Case Applications

Industrial and Manufacturing

- **Predictive Maintenance**: Detecting anomalies in machinery motion to predict failures before they occur.
- **Quality Control**: Monitoring the motion of assembly lines or robotic arms for precision assessment and deviation detection from the standard motion pattern.
- **Warehouse Logistics**: Managing and tracking the movement of goods with automated systems that classify different types of motion and detect anomalies in handling.

Healthcare

- **Patient Monitoring**: Detecting falls or abnormal movements in the elderly or those with mobility issues.
- **Rehabilitation**: Monitoring the progress of patients recovering from injuries by classifying motion patterns during physical therapy sessions.
- **Activity Recognition**: Classifying types of physical activity for fitness applications or patient monitoring.

Consumer Electronics

- **Gesture Control**: Interpreting specific motions to control devices, such as turning on lights with a hand wave.
- **Gaming**: Enhancing gaming experiences with motion-controlled inputs.

Transportation and Logistics

- **Vehicle Telematics**: Monitoring vehicle motion for unusual behavior such as hard braking, sharp turns, or accidents.
- **Cargo Monitoring**: Ensuring the integrity of goods during transport by detecting unusual movements that could indicate tampering or mishandling.

Smart Cities and Infrastructure

- **Structural Health Monitoring**: Detecting vibrations or movements within structures that could indicate potential failures or maintenance needs.
- **Traffic Management**: Analyzing the flow of pedestrians or vehicles to improve urban mobility and safety.

Security and Surveillance

- **Intruder Detection**: Detecting motion patterns typical of unauthorized access or other security breaches.
- **Wildlife Monitoring**: Detecting poachers or abnormal animal movements in protected areas.

Agriculture

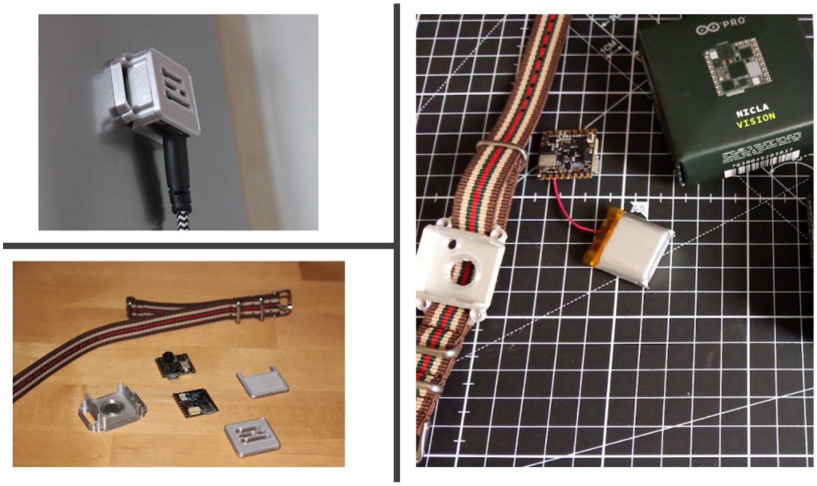
- **Equipment Monitoring**: Tracking the performance and usage of agricultural machinery.
- **Animal Behavior Analysis**: Monitoring livestock movements to detect behaviors indicating health issues or stress.

Environmental Monitoring

- **Seismic Activity**: Detecting irregular motion patterns that precede earthquakes or other geologically relevant events.
- **Oceanography**: Studying wave patterns or marine movements for research and safety purposes.

Nicla 3D case

For real applications, as some described before, we can add a case to our device, and Eoin Jordan, from Edge Impulse, developed a great wearable and machine health case for the Nicla range of boards. It works with a 10mm magnet, 2M screws, and a 16mm strap for human and machine health use case scenarios. Here is the link: Arduino Nicla Voice and Vision Wearable Case.



The applications for motion classification and anomaly detection are extensive, and the Arduino Nicla Vision is well-suited for scenarios where low power consumption and edge processing are advantageous. Its small form factor and efficiency in processing make it an ideal choice for deploying portable and remote applications where real-time processing is crucial and connectivity may be limited.

Resources

- Arduino Code
- Edge Impulse Spectral Features Block Colab Notebook
- Edge Impulse Project

XIAO ESP32S3

These labs provide a unique opportunity to gain practical experience with machine learning (ML) systems. Unlike working with large models requiring data center-scale resources, these exercises allow you to directly interact with hardware and software using TinyML. This hands-on approach gives you a tangible understanding of the challenges and opportunities in deploying AI, albeit at a tiny scale. However, the principles are largely the same as what you would encounter when working with larger systems.

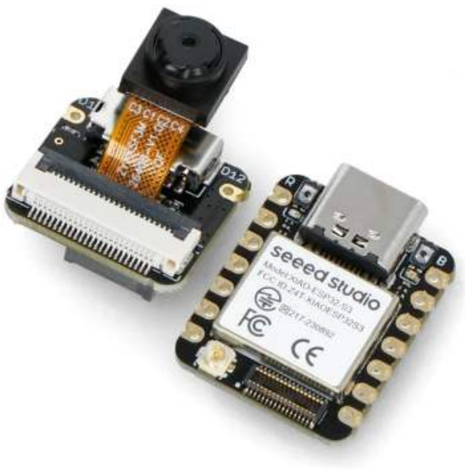


Figure 20.8: XIAO ESP32S3 Sense. Source: SEEED Studio

Pre-requisites

- **XIAO ESP32S3 Sense Board**: Ensure you have the XIAO ESP32S3 Sense Board.
- **USB-C Cable**: This is for connecting the board to your computer.
- **Network**: With internet access for downloading necessary software.
- **SD Card and an SD card Adapter**: This saves audio and images (optional).

Setup

- Setup XIAO ESP32S3

Exercises

Modality	Task	Description	Link
Vision	Image Classification	Learn to classify images	Link
Vision	Object Detection	Implement object detection	Link
Sound	Keyword Spotting	Explore voice recognition systems	Link
IMU	Motion Classification and Anomaly Detection	Classify motion data and detect anomalies	Link

Setup

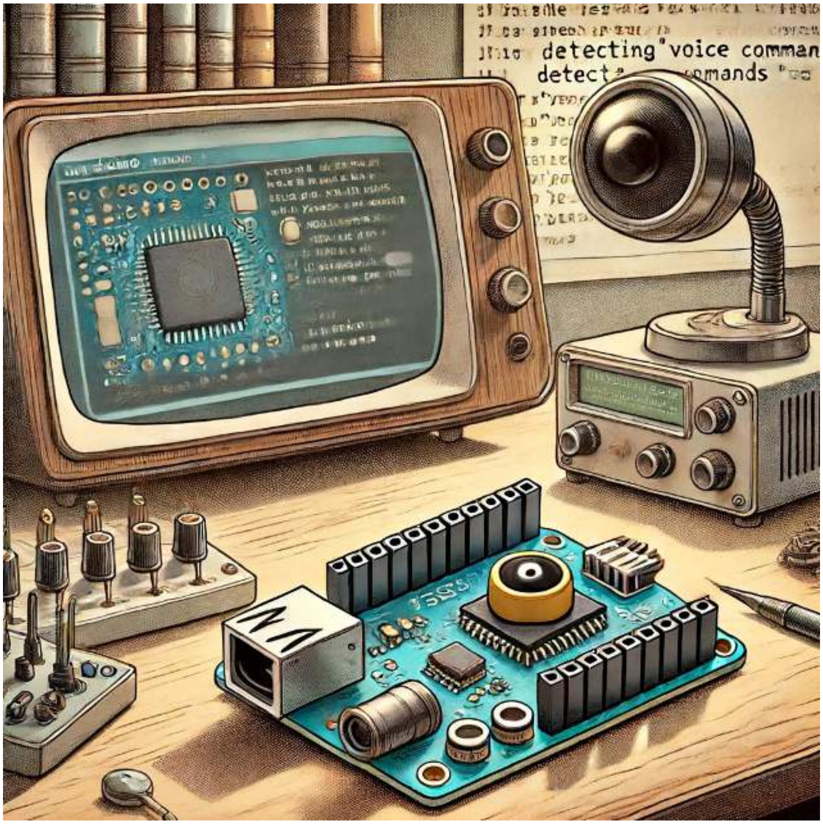


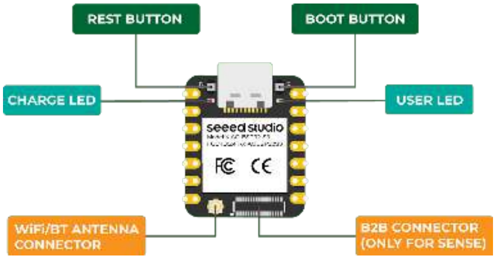
Figure 20.9: *DALL·E prompt - 1950s cartoon-style drawing of a XIAO ESP32S3 board with a distinctive camera module, as shown in the image provided. The board is placed on a classic lab table with various sensors, including a microphone. Behind the board, a vintage computer screen displays the Arduino IDE in muted colors, with code focusing on LED pin setups and machine learning inference for voice commands. The Serial Monitor on the IDE showcases outputs detecting voice commands like 'yes' and 'no'. The scene merges the retro charm of mid-century labs with modern electronics.*

Overview

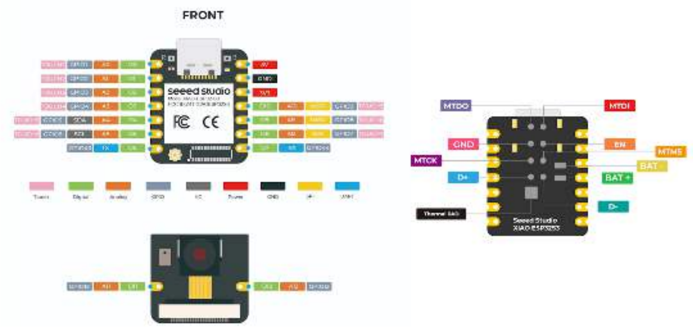
The XIAO ESP32S3 Sense is Seeed Studio's affordable development board, which integrates a camera sensor, digital microphone, and SD card support. Combining embedded ML computing power and photography capability, this development board is a great tool to start with TinyML (intelligent voice and vision AI).

XIAO ESP32S3 Sense Main Features

- **Powerful MCU Board**: Incorporate the ESP32S3 32-bit, dual-core, Xtensa processor chip operating up to 240 MHz, mounted multiple development ports, Arduino / MicroPython supported
- **Advanced Functionality**: Detachable OV2640 camera sensor for 1600 * 1200 resolution, compatible with OV5640 camera sensor, integrating an additional digital microphone
- **Elaborate Power Design**: Lithium battery charge management capability offers four power consumption models, which allows for deep sleep mode with power consumption as low as 14 μA
- **Great Memory for more Possibilities**: Offer 8 MB PSRAM and 8 MB FLASH, supporting SD card slot for external 32 GB FAT memory
- **Outstanding RF performance**: Support 2.4 GHz Wi-Fi and BLE dual wireless communication, support 100m+ remote communication when connected with U.FL antenna
- **Thumb-sized Compact Design**: 21 × 17.5 mm, adopting the classic form factor of XIAO, suitable for space-limited projects like wearable devices



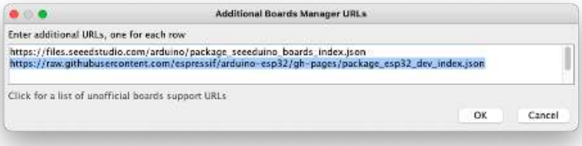
Below is the general board pinout:



For more details, please refer to the Seeed Studio WiKi page: https://wiki.seeedstudio.com/xiao_esp32s3_getting_started/

Installing the XIAO ESP32S3 Sense on Arduino IDE

On Arduino IDE, navigate to **File > Preferences**, and fill in the URL:
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_dev_index.json
on the field ==> **Additional Boards Manager URLs**

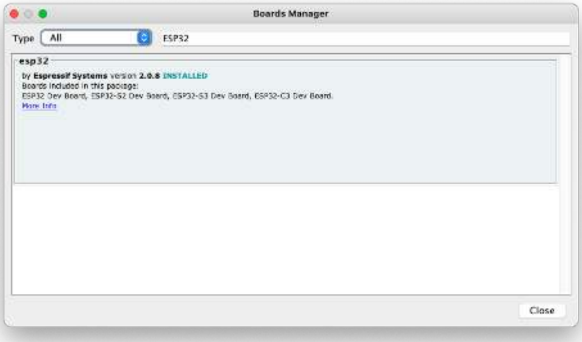


Enter additional URLs, one for each row

```
https://files.seeedstudio.com/arduino/package_seeeduino_boards_index.json
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_dev_index.json
```

Click for a list of unofficial boards support URLs

Next, open boards manager. Go to **Tools** > **Board** > **Boards Manager...** and enter with *esp32*. Select and install the most updated and stable package (avoid *alpha* versions):



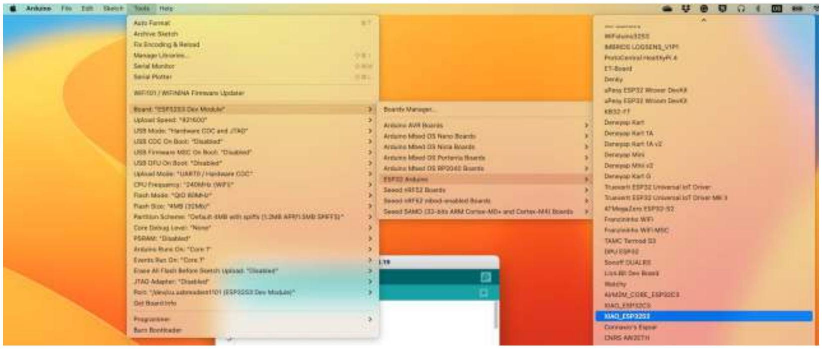
Type: All   ESP32

esp32
by **Espressif Systems** version **2.0.8** INSTALLED
Boards included in this package:
ESP32 Dev Board, ESP32-S2 Dev Board, ESP32-S3 Dev Board, ESP32-C3 Dev Board.
More Info

Attention

Alpha versions (for example, 3.x-alpha) do not work correctly with the XIAO and Edge Impulse. Use the last stable version (for example, 2.0.11) instead.

On **Tools**, select the Board (**XIAO ESP32S3**):


Arduino
File
Edit
Sketch
Tools
Help
Auto Format
Archive Sketch
Fix Encoding & Reload
Manage Libraries...
Serial Monitor
Serial Plotter
WiFi101 / WiFiNINA Firmware Updater
Board: "ESP32S3 Dev Module"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
USB CDC On Boot: "Disabled"
USB Firmware MSC On Boot: "Disabled"
USB DFU On Boot: "Disabled"
Upload Mode: "UART0 / Hardware CDC"
CPU Frequency: "240MHz (WiFi)"
Flash Mode: "QIO 80MHz"
Flash Size: "4MB (32Mb)"
Partition Scheme: "Default 4MB with spiffs (1.2MB APP/1.5MB SPIFFS)"
Core Debug Level: "None"
PSRAM: "Disabled"
Arduino Runs On: "Core 1"
Events Run On: "Core 1"
Erase All Flash Before Sketch Upload: "Disabled"
JTAG Adapter: "Disabled"
Port: "/dev/cu.usbmodem1101 (ESP32S3 Dev Module)"
Get Board Info
Programmer
Burn Bootloader
Boards Manager...
Arduino AVR Boards
Arduino Mbed OS Nano Boards
Arduino Mbed OS Nicla Boards
Arduino Mbed OS Portenta Boards
Arduino Mbed OS RP2040 Boards
ESP32 Arduino
Seeed nRF52 Boards
Seeed nRF52 mbed-enabled Boards
Seeed SAMD (32-bits ARM Cortex-M0+ and Cortex-M4) Boards
ET-Board
Denky
KB32-FT
Deneyap Kart
Deneyap Kart 1A
Deneyap Kart 1A v2
Deneyap Mini
Deneyap Mini v2
Deneyap Kart G
Franzininho WiFi
Franzininho WiFi MSC
TAMC Termod S3
Watchy
XIAO_ESP32C3
XIAO_ESP32S3
CNRS AW2ETH

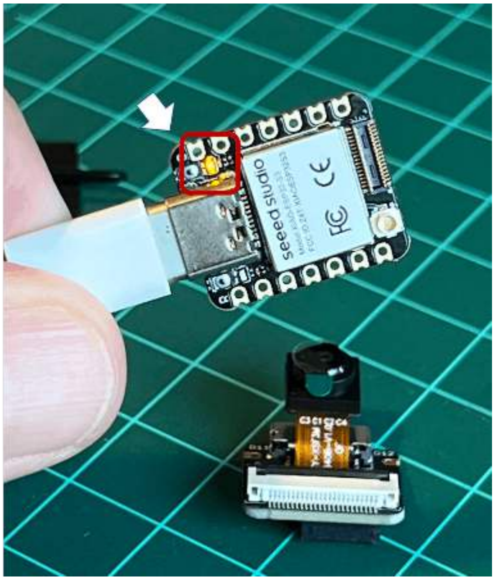
Last but not least, choose the **Port** where the ESP32S3 is connected. That is it! The device should be OK. Let's do some tests.

Testing the board with BLINK

The XIAO ESP32S3 Sense has a built-in LED that is connected to GPIO21. So, you can run the blink sketch as it is (using the `LED_BUILTIN` Arduino constant) or by changing the Blink sketch accordingly:

```
#define LED_BUILT_IN 21

void setup() {
  pinMode(LED_BUILT_IN, OUTPUT); // Set the pin as output
}

// Remember that the pin work with inverted logic
// LOW to Turn on and HIGH to turn off
void loop() {
  digitalWrite(LED_BUILT_IN, LOW); //Turn on
  delay (1000); //Wait 1 sec
  digitalWrite(LED_BUILT_IN, HIGH); //Turn off
  delay (1000); //Wait 1 sec
}
```

Note that the pins work with inverted logic: LOW to Turn on and HIGH to turn off.



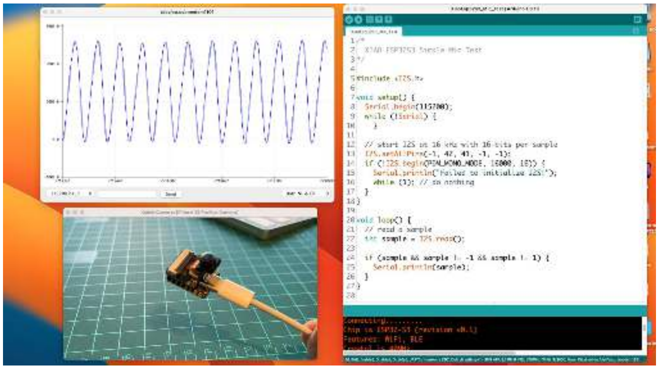
Connecting Sense module (Expansion Board)

When purchased, the expansion board is separated from the main board, but installing the expansion board is very simple. You need to align the connector on the expansion board with the B2B connector on the XIAO ESP32S3, press it hard, and when you hear a "click," the installation is complete.

As commented in the introduction, the expansion board, or the "sense" part of the device, has a 1600 × 1200 OV2640 camera, an SD card slot, and a digital microphone.

Microphone Test

Let's start with sound detection. Go to the GitHub project and download the sketch: XIAOEsp2s3_Mic_Test and run it on the Arduino IDE:



```
/*
  XIAO ESP32S3 Simple Mic Test
*/

#include <I2S.h>

void setup() {
  Serial.begin(115200);
  while (!Serial) {
    }

  // start I2S at 16 kHz with 16-bits per sample
  I2S.setAllPins(-1, 42, 41, -1, -1);
  if (!I2S.begin(PDM_MONO_MODE, 16000, 16)) {
    Serial.println("Failed to initialize I2S!");
    while (1); // do nothing
  }
}

void loop() {
  // read a sample
  int sample = I2S.read();

  if (sample && sample != -1 && sample != 1) {
    Serial.println(sample);
  }
}
```

```
Connecting........
Chip is ESP32-S3 (revision v0.1)
Features: WiFi, BLE
```

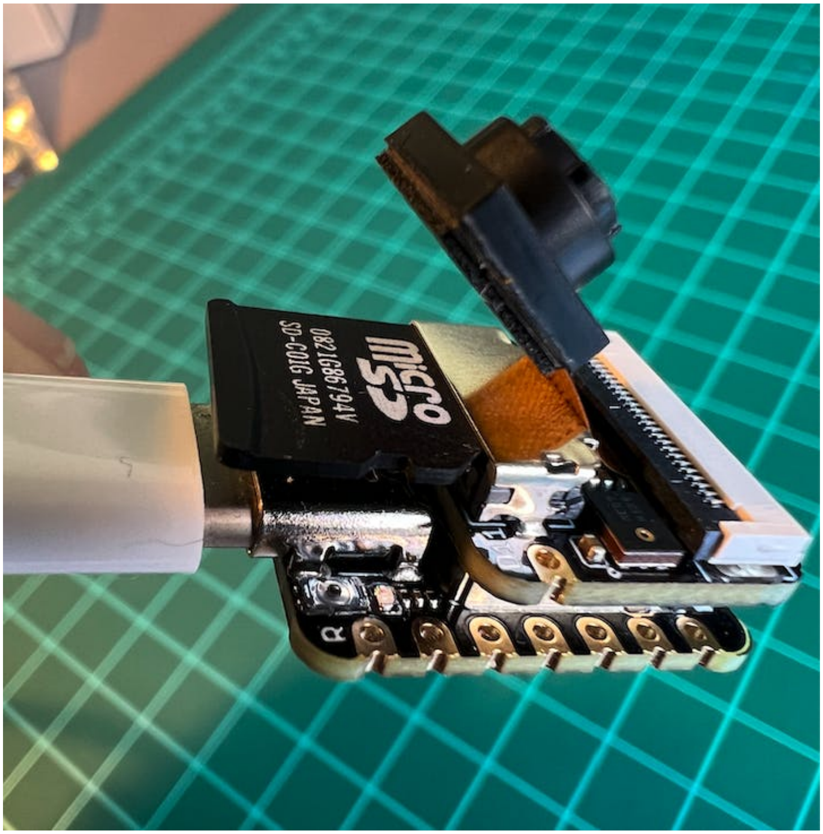
When producing sound, you can verify it on the Serial Plotter.

Save recorded sound (.wav audio files) to a microSD card.

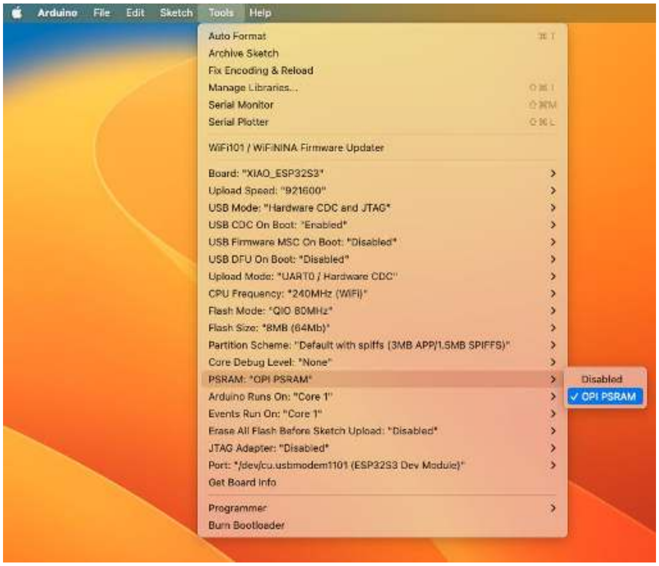
Now, the onboard SD Card reader can save .wav audio files. To do that, we need to habilitate the XIAO PSRAM.

> ESP32-S3 has only a few hundred kilobytes of internal RAM on the MCU chip. This can be insufficient for some purposes, so up to 16 MB of external PSRAM (pseudo-static RAM) can be connected with the SPI flash chip. The external memory is incorporated in the memory map and, with certain restrictions, is usable in the same way as internal data RAM.

For a start, Insert the SD Card on the XIAO as shown in the photo below (the SD Card should be formatted to **FAT32**).

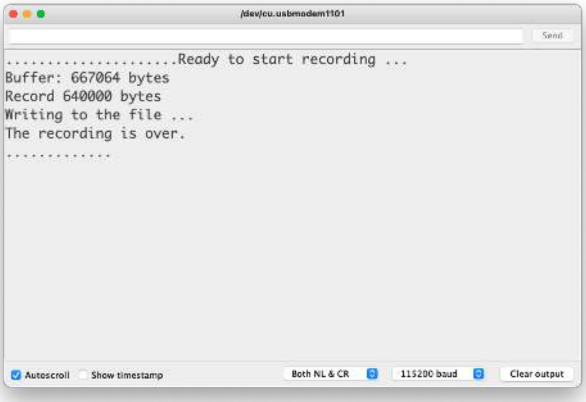


- Download the sketch Wav_Record, which you can find on GitHub.
- To execute the code (Wav Record), it is necessary to use the PSRAM function of the ESP-32 chip, so turn it on before uploading: Tools>PSRAM: "OPI PSRAM">OPI PSRAM



Board: "XIAO_ESP32S3"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
USB CDC On Boot: "Enabled"
USB Firmware MSC On Boot: "Disabled"
USB DFU On Boot: "Disabled"
Upload Mode: "UART0 / Hardware CDC"
CPU Frequency: "240MHz (WiFi)"
Flash Mode: "QIO 80MHz"
Flash Size: "8MB (64Mb)"
Partition Scheme: "Default with spiffs (3MB APP/1.5MB SPIFFS)"
Core Debug Level: "None"
PSRAM: "OPI PSRAM"
Disabled
✓ OPI PSRAM
Arduino Runs On: "Core 1"
Events Run On: "Core 1"
Erase All Flash Before Sketch Upload: "Disabled"
JTAG Adapter: "Disabled"
Port: "/dev/cu.usbmodem1101 (ESP32S3 Dev Module)"

- Run the code `Wav_Record.ino`
- This program is executed only once after the user turns on the serial monitor. It records for 20 seconds and saves the recording file to a microSD card as “arduino_rec.wav.”
- When the “.” is output every 1 second in the serial monitor, the program execution is finished, and you can play the recorded sound file with the help of a card reader.



```
.....................Ready to start recording ...
Buffer: 667064 bytes
Record 640000 bytes
Writing to the file ...
The recording is over.
.............
```

The sound quality is excellent!

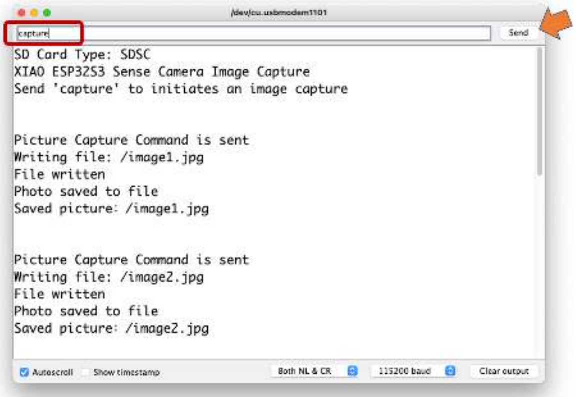
The explanation of how the code works is beyond the scope of this tutorial, but you can find an excellent description on the wiki page.

Testing the Camera

To test the camera, you should download the folder take_photos_command from GitHub. The folder contains the sketch (.ino) and two .h files with camera details.

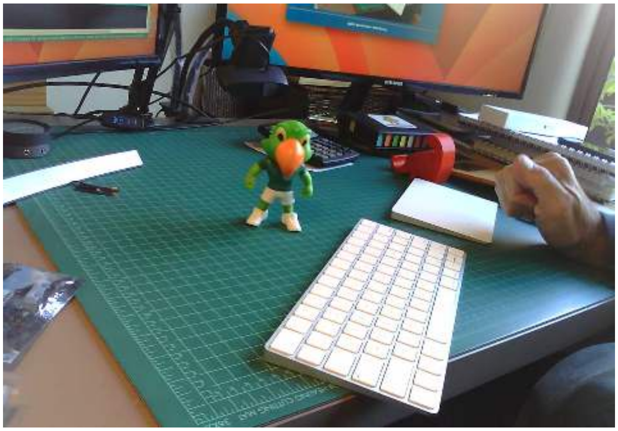
- Run the code: `take_photos_command.ino`. Open the Serial Monitor and send the command `capture` to capture and save the image on the SD Card:

  Verify that `[Both NL & CR]` are selected on Serial Monitor.



```
SD Card Type: SDSC
XIAO ESP32S3 Sense Camera Image Capture
Send 'capture' to initiates an image capture


Picture Capture Command is sent
Writing file: /image1.jpg
File written
Photo saved to file
Saved picture: /image1.jpg


Picture Capture Command is sent
Writing file: /image2.jpg
File written
Photo saved to file
Saved picture: /image2.jpg
```

Here is an example of a taken photo:

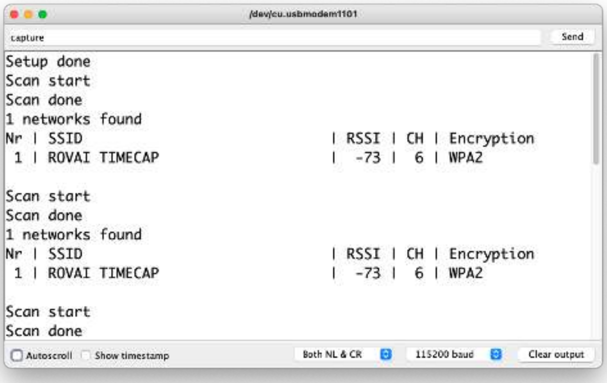


Testing WiFi

One of the XIAO ESP32S3's differentiators is its WiFi capability. So, let's test its radio by scanning the Wi-Fi networks around it. You can do this by running one of the code examples on the board.

Go to Arduino IDE Examples and look for **WiFI ==> WiFIScan**

You should see the Wi-Fi networks (SSIDs and RSSIs) within your device's range on the serial monitor. Here is what I got in the lab:



```
Setup done
Scan start
Scan done
1 networks found
Nr | SSID                             | RSSI | CH | Encryption
 1 | ROVAI TIMECAP                    |  -73 |  6 | WPA2

Scan start
Scan done
1 networks found
Nr | SSID                             | RSSI | CH | Encryption
 1 | ROVAI TIMECAP                    |  -73 |  6 | WPA2

Scan start
Scan done
```

Simple WiFi Server (Turning LED ON/OFF)

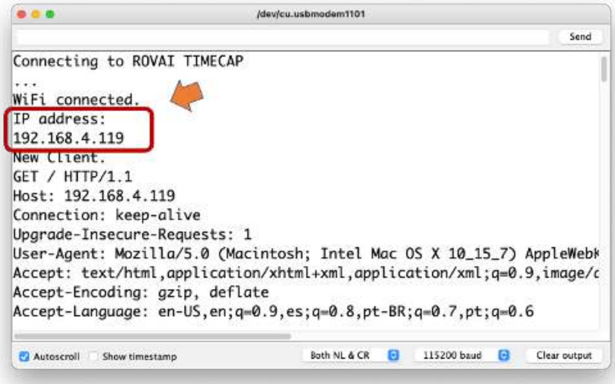
Let's test the device's capability to behave as a WiFi Server. We will host a simple page on the device that sends commands to turn the XIAO built-in LED ON and OFF.

Like before, go to GitHub to download the folder using the sketch SimpleWiFiServer.

Before running the sketch, you should enter your network credentials:

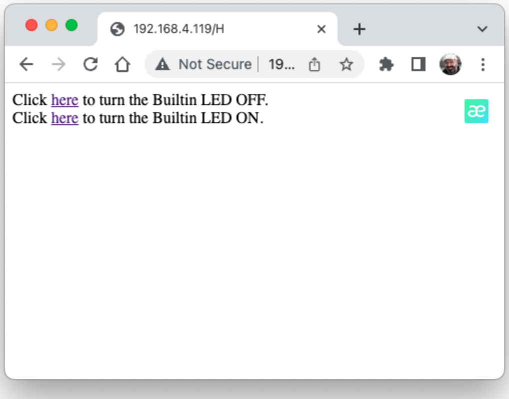
```
const char* ssid     = "Your credentials here";
const char* password = "Your credentials here";
```

You can monitor how your server is working with the Serial Monitor.



```
Connecting to ROVAI TIMECAP
...
WiFi connected.
IP address:
192.168.4.119
New Client.
GET / HTTP/1.1
Host: 192.168.4.119
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebk
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/c
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,es;q=0.8,pt-BR;q=0.7,pt;q=0.6
```

Take the IP address and enter it on your browser:



Click here to turn the Builtin LED OFF.
Click here to turn the Builtin LED ON.

You will see a page with links that can turn the built-in LED of your XIAO ON and OFF.

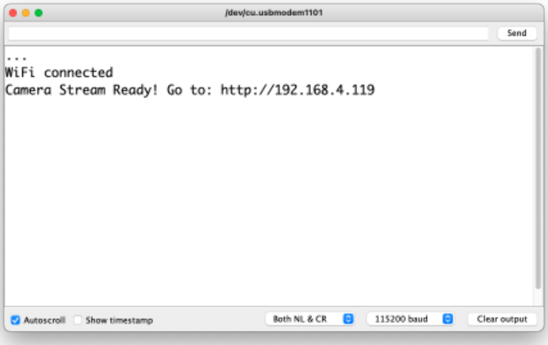
Streaming video to Web

Now that you know that you can send commands from the webpage to your device, let's do the reverse. Let's take the image captured by the camera and stream it to a webpage:

Download from GitHub the folder that contains the code: XIAO-ESP32S3-Streeming_Video.ino.

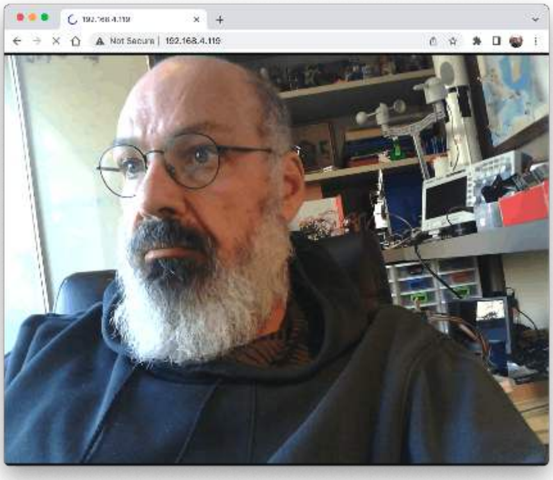
> Remember that the folder contains the.ino file and a couple of .h files necessary to handle the camera.

Enter your credentials and run the sketch. On the Serial monitor, you can find the page address to enter in your browser:



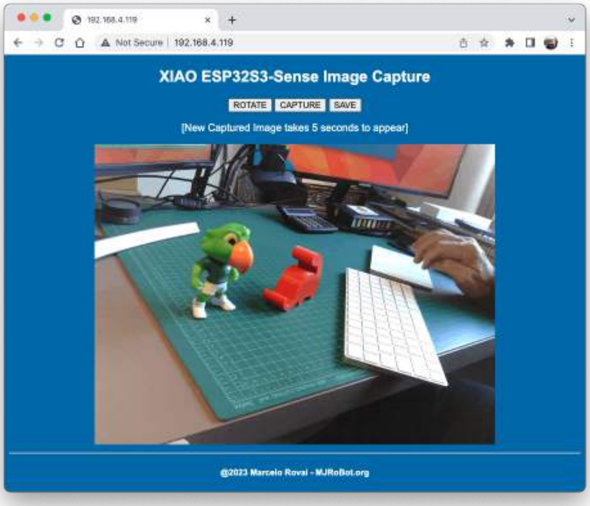
```
...
WiFi connected
Camera Stream Ready! Go to: http://192.168.4.119
```

Open the page on your browser (wait a few seconds to start the streaming). That's it.



Streamlining what your camera is "seen" can be important when you position it to capture a dataset for an ML project (for example, using the code "take_phots_commands.ino".

Of course, we can do both things simultaneously: show what the camera sees on the page and send a command to capture and save the image on the SD card. For that, you can use the code Camera_HTTP_Server_STA, which can be downloaded from GitHub.

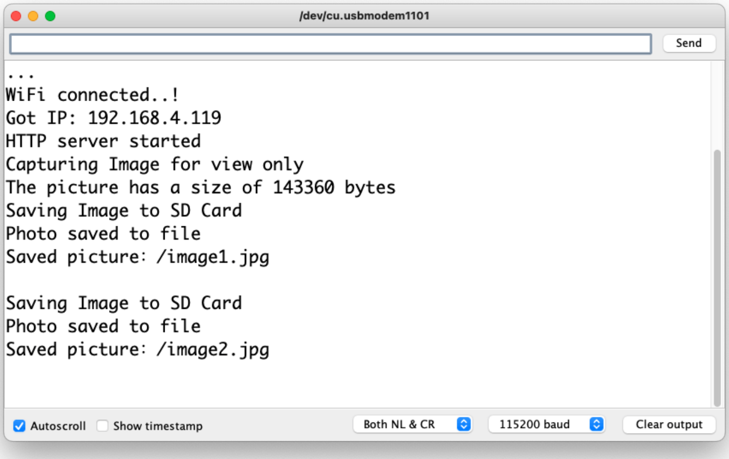

XIAO ESP32S3-Sense Image Capture

[New Captured Image takes 5 seconds to appear]

@2023 Marcelo Rovai - MJRoBot.org

The program will do the following tasks:

- Set the camera to JPEG output mode.
- Create a web page (for example ==> http://192.168.4.119//). The correct address will be displayed on the Serial Monitor.
- If server.on ("/capture", HTTP_GET, serverCapture), the program takes a photo and sends it to the Web.
- It is possible to rotate the image on webPage using the button [ROTATE]
- The command [CAPTURE] only will preview the image on the webpage, showing its size on the Serial Monitor
- The `[SAVE]` command will save an image on the SD Card and show the image on the browser.
- Saved images will follow a sequential naming (image1.jpg, image2.jpg.



```
...
WiFi connected..!
Got IP: 192.168.4.119
HTTP server started
Capturing Image for view only
The picture has a size of 143360 bytes
Saving Image to SD Card
Photo saved to file
Saved picture: /image1.jpg

Saving Image to SD Card
Photo saved to file
Saved picture: /image2.jpg
```

> This program can capture an image dataset with an image classification project.

Inspect the code; it will be easier to understand how the camera works. This code was developed based on the great Rui Santos Tutorial ESP32-CAM Take Photo and Display in Web Server, which I invite all of you to visit.

Using the CameraWebServer

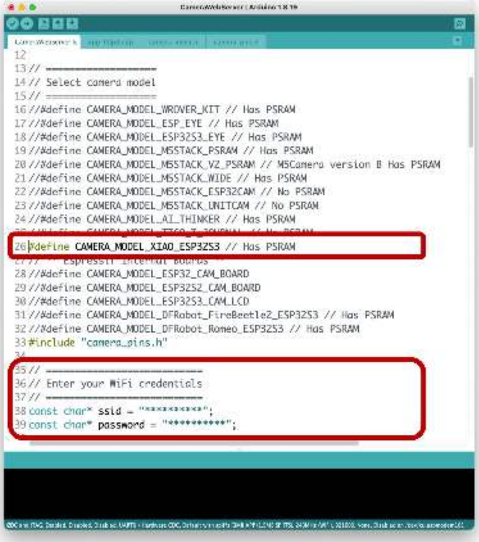
In the Arduino IDE, go to `File > Examples > ESP32 > Camera`, and select `CameraWebServer`

You also should comment on all cameras' models, except the XIAO model pins:

`#define CAMERA_MODEL_XIAO_ESP32S3 // Has PSRAM`

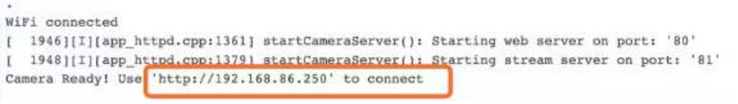
Do not forget the `Tools` to enable the PSRAM.

Enter your wifi credentials and upload the code to the device:



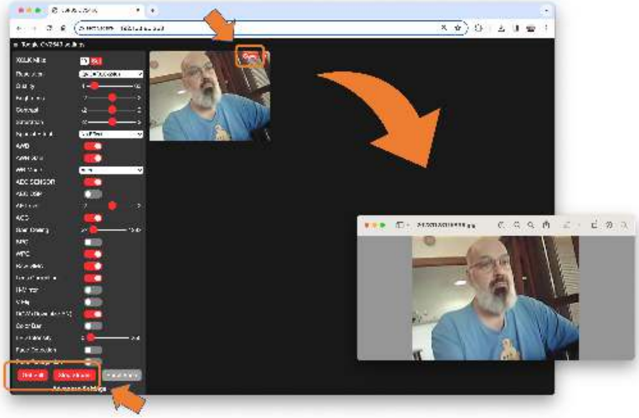
```
// ===================
// Select camera model
// ===================
//#define CAMERA_MODEL_WROVER_KIT // Has PSRAM
//#define CAMERA_MODEL_ESP_EYE // Has PSRAM
//#define CAMERA_MODEL_ESP32S3_EYE // Has PSRAM
//#define CAMERA_MODEL_M5STACK_PSRAM // Has PSRAM
//#define CAMERA_MODEL_M5STACK_V2_PSRAM // M5Camera version B Has PSRAM
//#define CAMERA_MODEL_M5STACK_WIDE // Has PSRAM
//#define CAMERA_MODEL_M5STACK_ESP32CAM // No PSRAM
//#define CAMERA_MODEL_M5STACK_UNITCAM // No PSRAM
//#define CAMERA_MODEL_AI_THINKER // Has PSRAM
//#define CAMERA_MODEL_TTGO_T_JOURNAL // No PSRAM
#define CAMERA_MODEL_XIAO_ESP32S3 // Has PSRAM
// ** Espressif Internal Boards **
//#define CAMERA_MODEL_ESP32_CAM_BOARD
//#define CAMERA_MODEL_ESP32S2_CAM_BOARD
//#define CAMERA_MODEL_ESP32S3_CAM_LCD
//#define CAMERA_MODEL_DFRobot_FireBeetle2_ESP32S3 // Has PSRAM
//#define CAMERA_MODEL_DFRobot_Romeo_ESP32S3 // Has PSRAM
#include "camera_pins.h"

// ===========================
// Enter your WiFi credentials
// ===========================
const char* ssid = "**********";
const char* password = "**********";
```

If the code is executed correctly, you should see the address on the Serial Monitor:



```
.
WiFi connected
[  1946][I][app_httpd.cpp:1361] startCameraServer(): Starting web server on port: '80'
[  1948][I][app_httpd.cpp:1379] startCameraServer(): Starting stream server on port: '81'
Camera Ready! Use 'http://192.168.86.250' to connect
```

Copy the address on your browser and wait for the page to be uploaded. Select the camera resolution (for example, QVGA) and select [START STREAM]. Wait for a few seconds/minutes, depending on your connection. Using the [Save] button, you can save an image to your computer download area.



That's it! You can save the images directly on your computer for use on projects.

Conclusion

The XIAO ESP32S3 Sense is flexible, inexpensive, and easy to program. With 8 MB of RAM, memory is not an issue, and the device can handle many post-processing tasks, including communication.

You will find the last version of the codeon the GitHub repository: XIAO-ESP32S3-Sense.

Resources

- XIAO ESP32S3 Code

Image Classification

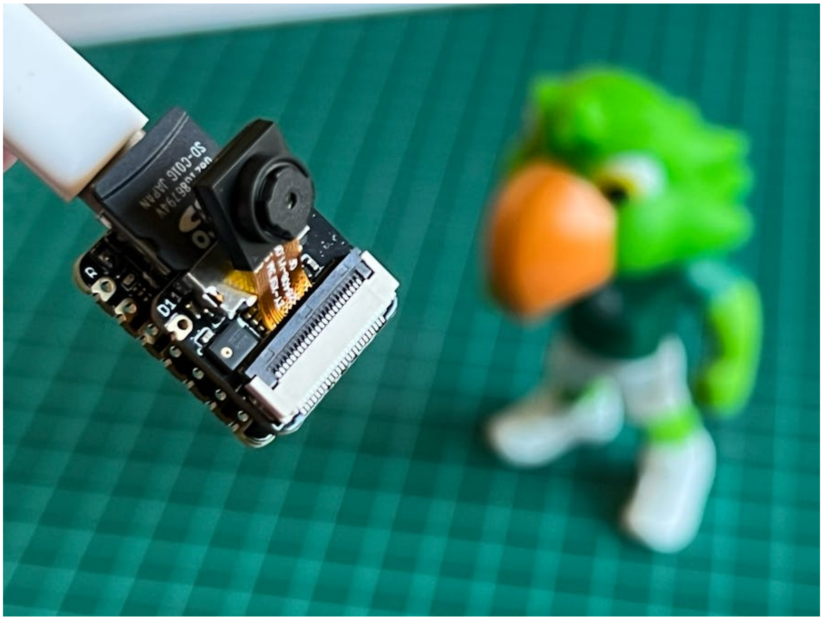
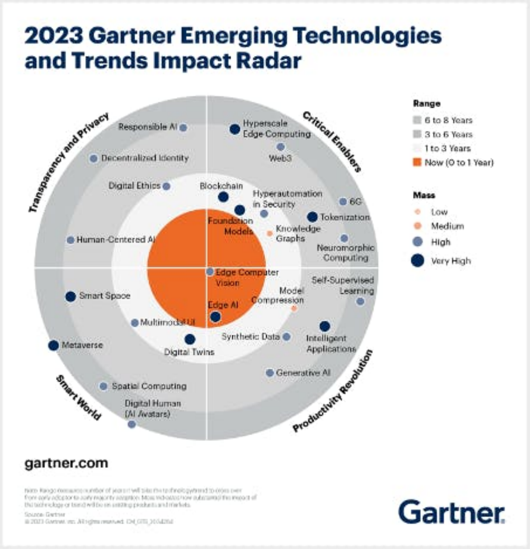


Figure 20.10: *Image by Marcelo Rovai*

Overview

More and more, we are facing an artificial intelligence (AI) revolution where, as stated by Gartner, **Edge AI** has a very high impact potential, and **it is for now**!



At the forefront of the Emerging Technologies Radar is the universal language of Edge Computer Vision. When we look into Machine Learning (ML) applied to vision, the first concept that greets us is Image Classification, a kind of ML' Hello World ' that is both simple and profound!

The Seeed Studio XIAO ESP32S3 Sense is a powerful tool that combines camera and SD card support. With its embedded ML computing power and photography capability, it is an excellent starting point for exploring TinyML vision AI.

A TinyML Image Classification Project – Fruits versus Veggies

The whole idea of our project will be to train a model and proceed with inference on the XIAO ESP32S3 Sense. For training, we should find some data **(in fact, tons of data!)**.

But first of all, we need a goal! What do we want to classify?

With TinyML, a set of techniques associated with machine learning inference on embedded devices, we should limit the classification to three or four categories due to limitations (mainly memory). We will differentiate **apples** from **bananas** and **potatoes** (you can try other categories)**.**

So, let's find a specific dataset that includes images from those categories. Kaggle is a good start:

https://www.kaggle.com/kritikseth/fruit-and-vegetable-image-recognition

This dataset contains images of the following food items:

- **Fruits** –*banana, apple*, pear, grapes, orange, kiwi, watermelon, pomegranate, pineapple, mango.
- **Vegetables** – cucumber, carrot, capsicum, onion, *potato,* lemon, tomato, radish, beetroot, cabbage, lettuce, spinach, soybean, cauliflower, bell pepper, chili pepper, turnip, corn, sweetcorn, sweet potato, paprika, jalepeño, ginger, garlic, peas, eggplant.

Each category is split into the **train** (100 images), **test** (10 images), and **validation** (10 images).

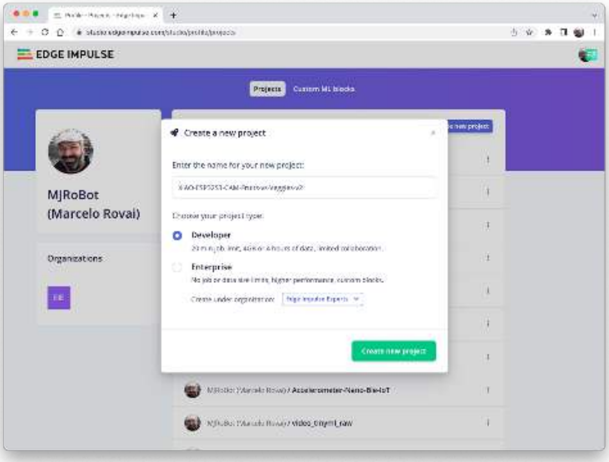
- Download the dataset from the Kaggle website and put it on your computer.

> Optionally, you can add some fresh photos of bananas, apples, and potatoes from your home kitchen, using, for example, the code discussed in the next setup lab.

Training the model with Edge Impulse Studio

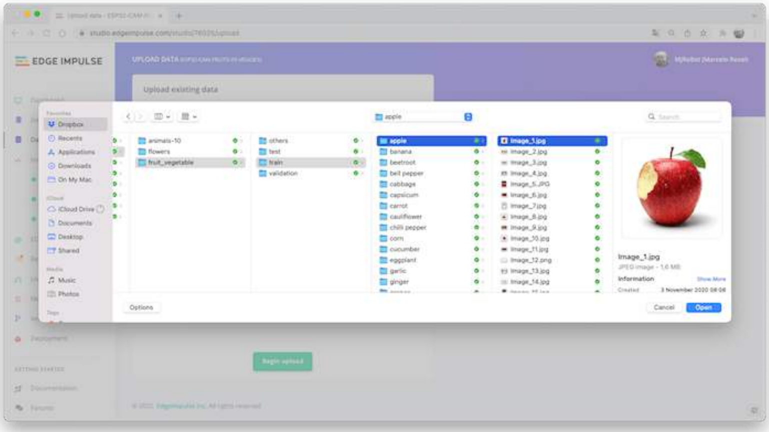
We will use the Edge Impulse Studio to train our model. As you may know, Edge Impulse is a leading development platform for machine learning on edge devices.

Enter your account credentials (or create a free account) at Edge Impulse. Next, create a new project:

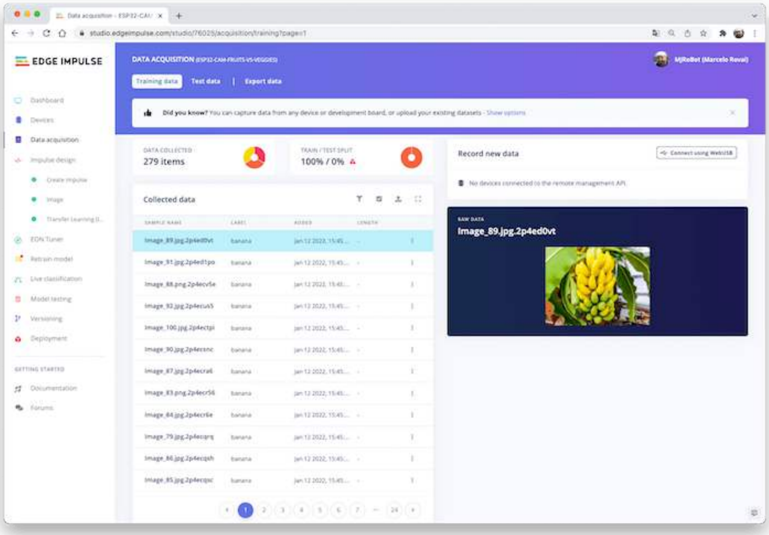

EDGE IMPULSE
Projects
Custom ML blocks
MJRoBot (Marcelo Rovai)
Organizations
EE
Create a new project
Enter the name for your new project:
XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v2
Choose your project type:
● Developer
20 min job limit, 4GB or 4 hours of data, limited collaboration.
○ Enterprise
No job or data size limits, higher performance, custom blocks.
Create under organization: Edge Impulse Experts
Create new project
MJRoBot (Marcelo Rovai) / Accelerometer-Nano-Ble-IoT
MJRoBot (Marcelo Rovai) / video_tinyml_raw

Data Acquisition

Next, on the `UPLOAD DATA` section, upload from your computer the files from chosen categories:



It would be best if you now had your training dataset split into three classes of data:



DATA ACQUISITION (ESP32-CAM-FRUITS-VS-VEGGIES)

Did you know? You can capture data from any device or development board, or upload your existing datasets - Show options

DATA COLLECTED: 279 items

TRAIN / TEST SPLIT: 100% / 0%

Record new data

No devices connected to the remote management API.

Collected data

SAMPLE NAME	LABEL	ADDED	LENGTH
image_89.jpg.2p4ed0vt	banana	Jan 12 2022, 15:45...	-
image_91.jpg.2p4ed1po	banana	Jan 12 2022, 15:45...	-
image_88.png.2p4ecv5e	banana	Jan 12 2022, 15:45...	-
image_92.jpg.2p4ecus5	banana	Jan 12 2022, 15:45...	-
image_100.jpg.2p4ectpi	banana	Jan 12 2022, 15:45...	-
image_90.jpg.2p4ecsnc	banana	Jan 12 2022, 15:45...	-
image_87.jpg.2p4ecra6	banana	Jan 12 2022, 15:45...	-
image_83.png.2p4ecr56	banana	Jan 12 2022, 15:45...	-
image_84.jpg.2p4ecr6e	banana	Jan 12 2022, 15:45...	-
image_79.jpg.2p4ecqrq	banana	Jan 12 2022, 15:45...	-
image_86.jpg.2p4ecqsh	banana	Jan 12 2022, 15:45...	-
image_85.jpg.2p4ecqsc	banana	Jan 12 2022, 15:45...	-

RAW DATA

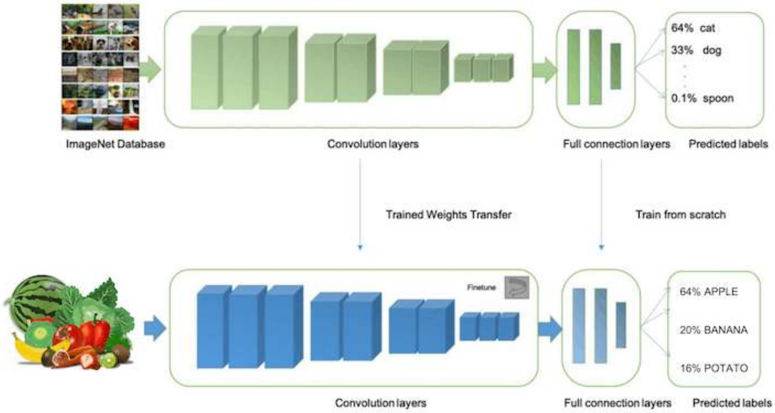
image_89.jpg.2p4ed0vt

You can upload extra data for further model testing or split the training data. I will leave it as it is to use the most data possible.

Impulse Design

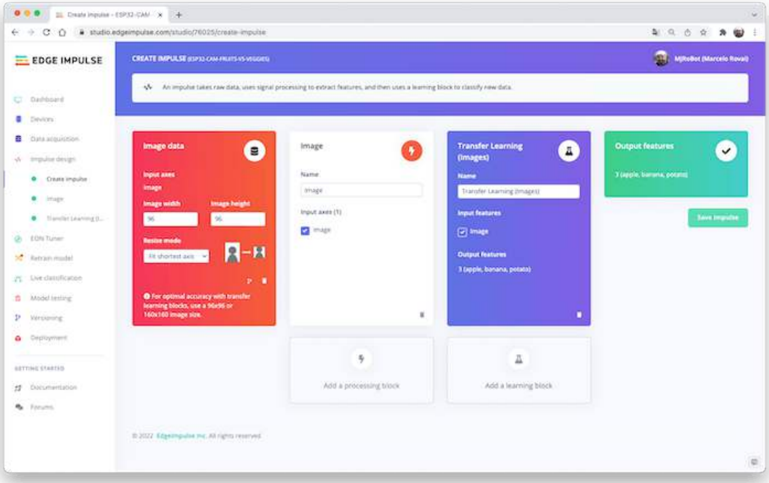
An impulse takes raw data (in this case, images), extracts features (resize pictures), and then uses a learning block to classify new data.

Classifying images is the most common use of deep learning, but a lot of data should be used to accomplish this task. We have around 90 images for each category. Is this number enough? Not at all! We will need thousands of images to "teach or model" to differentiate an apple from a banana. But, we can solve this issue by re-training a previously trained model with thousands of images. We call this technique "Transfer Learning" (TL).



With TL, we can fine-tune a pre-trained image classification model on our data, performing well even with relatively small image datasets (our case).

So, starting from the raw images, we will resize them (96×96) pixels and feed them to our Transfer Learning block:



CREATE IMPULSE (ESP32-CAM-FRUITS-VS-VEGGIES)

An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

Image data
Input axes
image
Image width: 96
Image height: 96
Resize mode: Fit shortest axis
For optimal accuracy with transfer learning blocks, use a 96x96 or 160x160 image size.

Image
Name: Image
Input axes (1)
☑ image

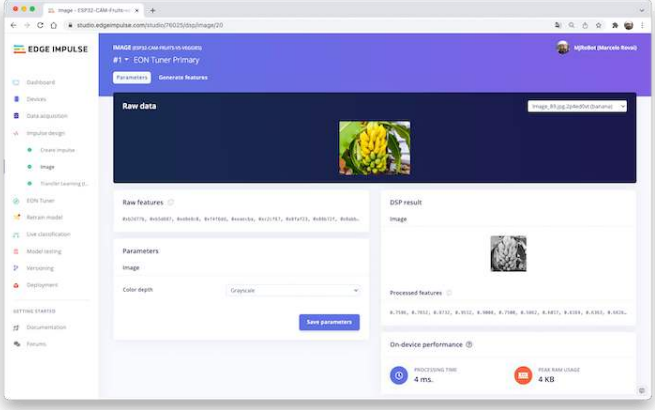
Transfer Learning (Images)
Name: Transfer Learning (Images)
Input features
☑ Image
Output features
3 (apple, banana, potato)

Output features
3 (apple, banana, potato)

© 2022 EdgeImpulse Inc. All rights reserved

Pre-processing (Feature Generation)

Besides resizing the images, we can change them to Grayscale or keep the actual RGB color depth. Let's start selecting `Grayscale`. Doing that, each one of our data samples will have dimension 9,216 features $(96 \times 96 \times 1)$. Keeping RGB, this dimension would be three times bigger. Working with Grayscale helps to reduce the amount of final memory needed for inference.



Color depth: Grayscale

Processing time: 4 ms.

Peak RAM usage: 4 KB

Remember to [Save parameters]. This will generate the features to be used in training.

Model Design

Transfer Learning

In 2007, Google introduced MobileNetV1, a family of general-purpose computer vision neural networks designed with mobile devices in mind to support classification, detection, and more. MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of various use cases.

Although the base MobileNet architecture is already tiny and has low latency, many times, a specific use case or application may require the model to be smaller and faster. MobileNet introduces a straightforward parameter α (alpha) called width multiplier to construct these smaller, less computationally expensive models. The role of the width multiplier α is to thin a network uniformly at each layer.

Edge Impulse Studio has **MobileNet V1 (96x96 images)** and **V2 (96x96 and 16x160 images)** available, with several different α values (from 0.05 to 1.0). For example, you will get the highest accuracy with V2, 160×160 images, and $\alpha = 1.0$. Of course, there is a trade-off. The higher the accuracy, the more memory (around 1.3 M RAM and 2.6 M ROM) will be needed to run the model, implying more latency.

The smaller footprint will be obtained at another extreme with **MobileNet V1** and $\alpha = 0.10$ (around 53.2 K RAM and 101 K ROM).

For this first pass, we will use **MobileNet V1** and $\alpha = 0.10$.

Training

Data Augmentation

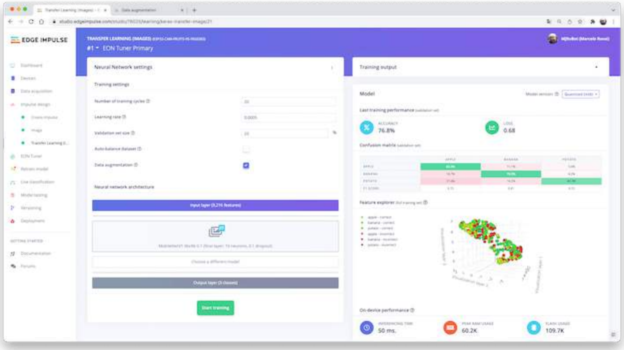
Another necessary technique to use with deep learning is **data augmentation**. Data augmentation is a method that can help improve the accuracy of machine learning models, creating additional artificial data. A data augmentation system makes small, random changes to your training data during the training process (such as flipping, cropping, or rotating the images).

Under the rood, here you can see how Edge Impulse implements a data Augmentation policy on your data:

```
# Implements the data augmentation policy
def augment_image(image, label):
    # Flips the image randomly
    image = tf.image.random_flip_left_right(image)

    # Increase the image size, then randomly crop it down to
    # the original dimensions
    resize_factor = random.uniform(1, 1.2)
    new_height = math.floor(resize_factor * INPUT_SHAPE[0])
    new_width = math.floor(resize_factor * INPUT_SHAPE[1])
    image = tf.image.resize_with_crop_or_pad(image, new_height,
                                             new_width)
    image = tf.image.random_crop(image, size=INPUT_SHAPE)

    # Vary the brightness of the image
    image = tf.image.random_brightness(image, max_delta=0.2)

    return image, label
```

Exposure to these variations during training can help prevent your model from taking shortcuts by “memorizing” superficial clues in your training data, meaning it may better reflect the deep underlying patterns in your dataset.

The final layer of our model will have 16 neurons with a 10% dropout for overfitting prevention. Here is the Training output:



EDGE IMPULSE

TRANSFER LEARNING (IMAGES)

#1 EON Tuner Primary

Neural Network settings

Training settings

Data augmentation ☑

Neural network architecture

Training output

Model

Last training performance

ACCURACY 76.8%

LOSS 0.68

Confusion matrix

Feature explorer

On-device performance

INFERENCING TIME 50 ms.

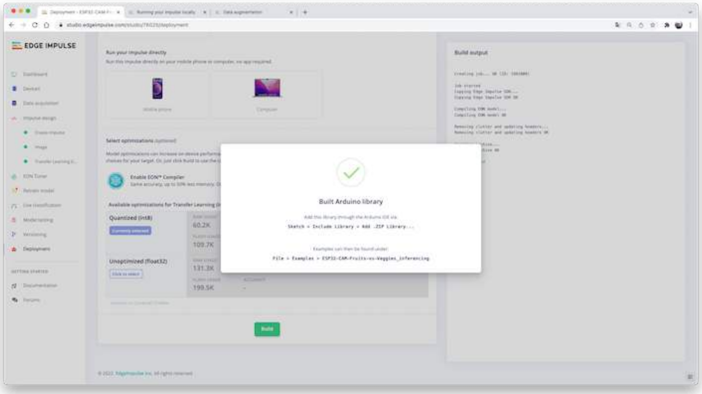
PEAK RAM USAGE 60.2K

FLASH USAGE 109.7K

The result could be better. The model reached around 77% accuracy, but the amount of RAM expected to be used during the inference is relatively tiny (about 60 KBytes), which is very good.

Deployment

The trained model will be deployed as a .zip Arduino library:



Built Arduino library

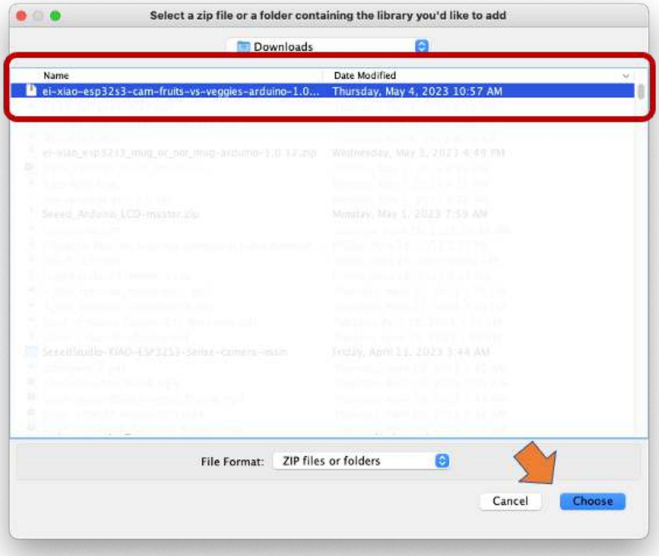
Add this library through the Arduino IDE via:

`Sketch > Include Library > Add .ZIP Library...`

Examples can then be found under:

`File > Examples > ESP32-CAM-Fruits-vs-Veggies_inferencing`

Open your Arduino IDE, and under **Sketch,** go to **Include Library** and **add.ZIP Library.** Please select the file you download from Edge Impulse Studio, and that's it!



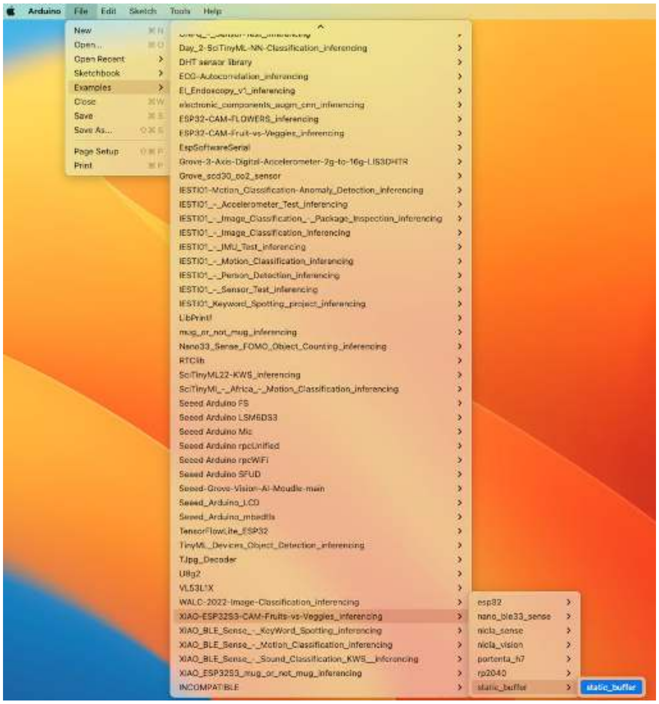
Select a zip file or a folder containing the library you'd like to add

Downloads

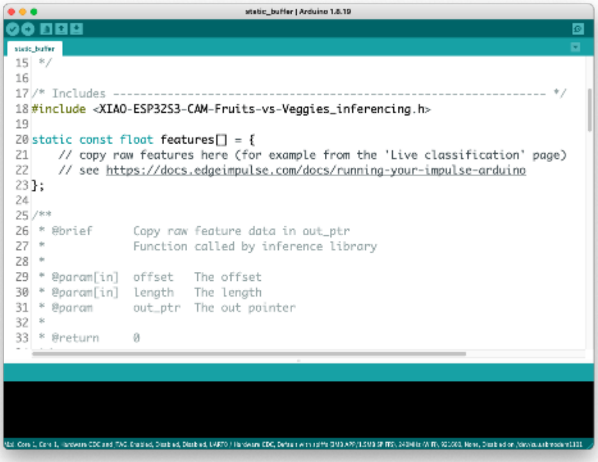
Name	Date Modified
ei-xiao-esp32s3-cam-fruits-vs-veggies-arduino-1.0...	Thursday, May 4, 2023 10:57 AM

File Format: ZIP files or folders

Under the **Examples** tab on Arduino IDE, you should find a sketch code under your project name.


Arduino
File
Edit
Sketch
Tools
Help
New
Open...
Open Recent
Sketchbook
Examples
Close
Save
Save As...
Page Setup
Print
Day_2-SciTinyML-NN-Classification_inferencing
DHT sensor library
ECG-Autocorrelation_inferencing
EI_Endoscopy_v1_inferencing
electronic_components_augm_cnn_inferencing
ESP32-CAM-FLOWERS_inferencing
ESP32-CAM-Fruit-vs-Veggies_inferencing
EspSoftwareSerial
Grove-3-Axis-Digital-Accelerometer-2g-to-16g-LIS3DHTR
Grove_scd30_co2_sensor
IESTI01-Motion_Classification-Anomaly_Detection_inferencing
IESTI01_-_Accelerometer_Test_inferencing
IESTI01_-_Image_Classification_-_Package_Inspection_inferencing
IESTI01_-_Image_Classification_inferencing
IESTI01_-_IMU_Test_inferencing
IESTI01_-_Motion_Classification_inferencing
IESTI01_-_Person_Detection_inferencing
IESTI01_-_Sensor_Test_inferencing
IESTI01_Keyword_Spotting_project_inferencing
LibPrintf
mug_or_not_mug_inferencing
Nano33_Sense_FOMO_Object_Counting_inferencing
RTClib
SciTinyML22-KWS_inferencing
SciTinyML_-_Africa_-_Motion_Classification_inferencing
Seeed Arduino FS
Seeed Arduino LSM6DS3
Seeed Arduino Mic
Seeed Arduino rpcUnified
Seeed Arduino rpcWiFi
Seeed Arduino SFUD
Seeed-Grove-Vision-AI-Moudle-main
Seeed_Arduino_LCD
Seeed_Arduino_mbedtls
TensorFlowLite_ESP32
TinyML_Devices_Object_Detection_inferencing
TJpg_Decoder
U8g2
VL53L1X
WALC-2022-Image-Classification_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing
XIAO_BLE_Sense_-_KeyWord_Spotting_inferencing
XIAO_BLE_Sense_-_Motion_Classification_inferencing
XIAO_BLE_Sense_-_Sound_Classification_KWS__inferencing
XIAO_ESP32S3_mug_or_not_mug_inferencing
INCOMPATIBLE
esp32
nano_ble33_sense
nicla_sense
nicla_vision
portenta_h7
rp2040
static_buffer
static_buffer

Open the Static Buffer example:



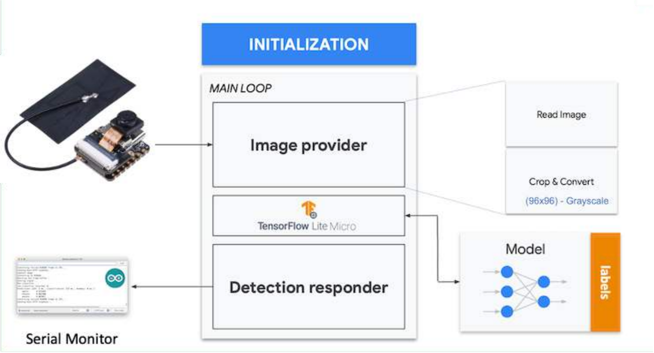
```
 */

/* Includes ---------------------------------------------------------------- */
#include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>

static const float features[] = {
    // copy raw features here (for example from the 'Live classification' page)
    // see https://docs.edgeimpulse.com/docs/running-your-impulse-arduino
};

/**
 * @brief      Copy raw feature data in out_ptr
 *             Function called by inference library
 *
 * @param[in]  offset   The offset
 * @param[in]  length   The length
 * @param      out_ptr  The out pointer
 *
 * @return     0
```

You can see that the first line of code is exactly the calling of a library with all the necessary stuff for running inference on your device.

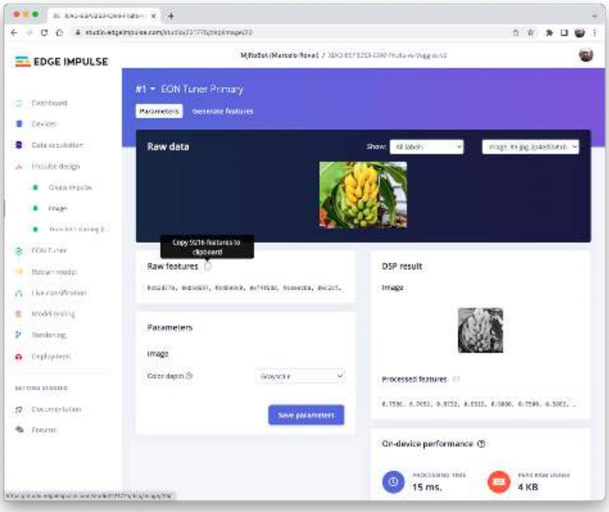
```
#include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>
```

Of course, this is a generic code (a "template") that only gets one sample of raw data (stored on the variable: features = {} and runs the classifier, doing the inference. The result is shown on the Serial Monitor.

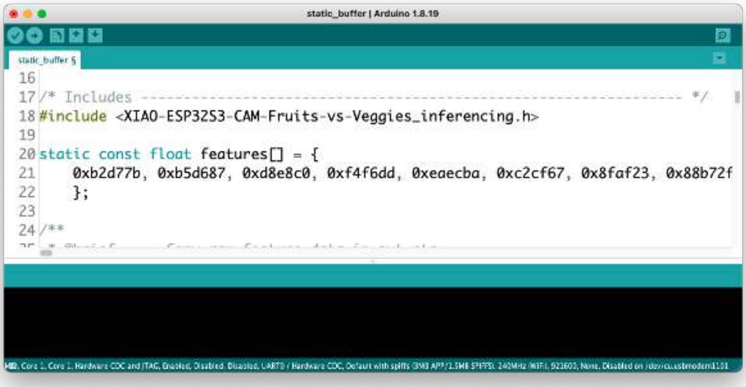
We should get the sample (image) from the camera and pre-process it (resizing to 96 × 96, converting to grayscale, and flatting it). This will be the input tensor of our model. The output tensor will be a vector with three values (labels), showing the probabilities of each one of the classes.



Returning to your project (Tab Image), copy one of the Raw Data Sample:



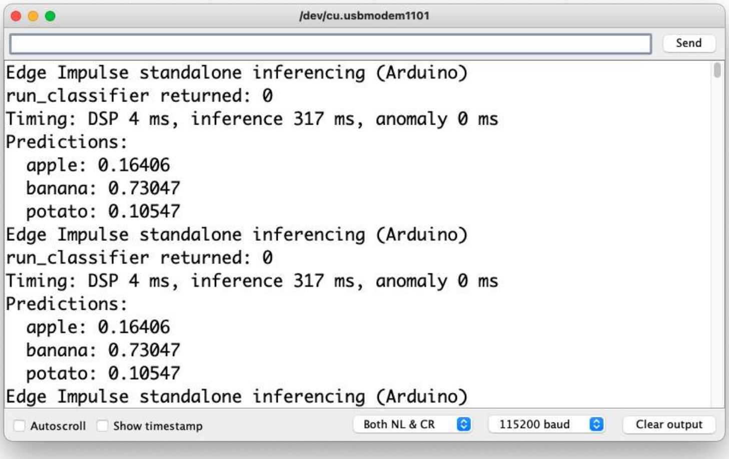
9,216 features will be copied to the clipboard. This is the input tensor (a flattened image of 96 × 96 × 1), in this case, bananas. Past this Input tensor on`features[] = {0xb2d77b, 0xb5d687, 0xd8e8c0, 0xeaecba, 0xc2cf67, ...}`


static_buffer | Arduino 1.8.19
static_buffer §
MB, Core 1, Core 1, Hardware CDC and JTAG, Enabled, Disabled, Disabled, UART0 / Hardware CDC, Default with spiffs (3MB APP/1.5MB SPIFFS), 240MHz (WiFi), 921600, None, Disabled on /dev/cu.usbmodem1101

```

/* Includes ---------------------------------------------------------------- */
#include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>

static const float features[] = {
    0xb2d77b, 0xb5d687, 0xd8e8c0, 0xf4f6dd, 0xeaecba, 0xc2cf67, 0x8faf23, 0x88b72f
    };

/**
```

Edge Impulse included the library ESP NN in its SDK, which contains optimized NN (Neural Network) functions for various Espressif chips, including the ESP32S3 (running at Arduino IDE).

When running the inference, you should get the highest score for "banana."

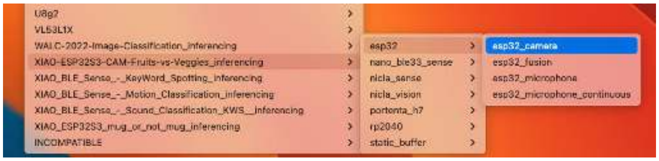


```
Edge Impulse standalone inferencing (Arduino)
run_classifier returned: 0
Timing: DSP 4 ms, inference 317 ms, anomaly 0 ms
Predictions:
  apple: 0.16406
  banana: 0.73047
  potato: 0.10547
Edge Impulse standalone inferencing (Arduino)
run_classifier returned: 0
Timing: DSP 4 ms, inference 317 ms, anomaly 0 ms
Predictions:
  apple: 0.16406
  banana: 0.73047
  potato: 0.10547
Edge Impulse standalone inferencing (Arduino)
```

Great news! Our device handles an inference, discovering that the input image is a banana. Also, note that the inference time was around 317 ms, resulting in a maximum of 3 fps if you tried to classify images from a video.

Now, we should incorporate the camera and classify images in real time.

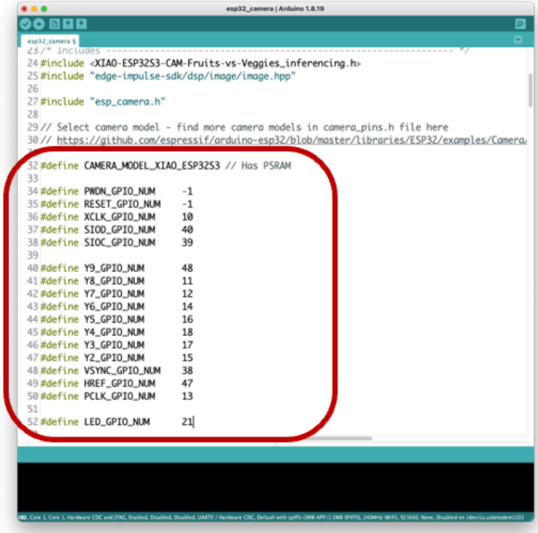
Go to the Arduino IDE Examples and download from your project the sketch `esp32_camera`:


U8g2
VL53L1X
WALC-2022-Image-Classification_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing
XIAO_BLE_Sense_-_KeyWord_Spotting_inferencing
XIAO_BLE_Sense_-_Motion_Classification_inferencing
XIAO_BLE_Sense_-_Sound_Classification_KWS__inferencing
XIAO_ESP32S3_mug_or_not_mug_inferencing
INCOMPATIBLE
esp32
nano_ble33_sense
nicla_sense
nicla_vision
portenta_h7
rp2040
static_buffer
esp32_camera
esp32_fusion
esp32_microphone
esp32_microphone_continuous

You should change lines 32 to 75, which define the camera model and pins, using the data related to our model. Copy and paste the below lines, replacing the lines 32-75:

```
#define PWDN_GPIO_NUM     -1
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM     10
#define SIOD_GPIO_NUM     40
#define SIOC_GPIO_NUM     39
#define Y9_GPIO_NUM       48
#define Y8_GPIO_NUM       11
#define Y7_GPIO_NUM       12
#define Y6_GPIO_NUM       14
#define Y5_GPIO_NUM       16
#define Y4_GPIO_NUM       18
#define Y3_GPIO_NUM       17
#define Y2_GPIO_NUM       15
#define VSYNC_GPIO_NUM    38
#define HREF_GPIO_NUM     47
#define PCLK_GPIO_NUM     13
```

Here you can see the resulting code:



```
23 /* Includes ---------------------------------------------------------------- */
24 #include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>
25 #include "edge-impulse-sdk/dsp/image/image.hpp"
26
27 #include "esp_camera.h"
28
29 // Select camera model - find more camera models in camera_pins.h file here
30 // https://github.com/espressif/arduino-esp32/blob/master/libraries/ESP32/examples/Camera/
31
32 #define CAMERA_MODEL_XIAO_ESP32S3 // Has PSRAM
33
34 #define PWDN_GPIO_NUM     -1
35 #define RESET_GPIO_NUM    -1
36 #define XCLK_GPIO_NUM     10
37 #define SIOD_GPIO_NUM     40
38 #define SIOC_GPIO_NUM     39
39
40 #define Y9_GPIO_NUM       48
41 #define Y8_GPIO_NUM       11
42 #define Y7_GPIO_NUM       12
43 #define Y6_GPIO_NUM       14
44 #define Y5_GPIO_NUM       16
45 #define Y4_GPIO_NUM       18
46 #define Y3_GPIO_NUM       17
47 #define Y2_GPIO_NUM       15
48 #define VSYNC_GPIO_NUM    38
49 #define HREF_GPIO_NUM     47
50 #define PCLK_GPIO_NUM     13
51
52 #define LED_GPIO_NUM      21
```

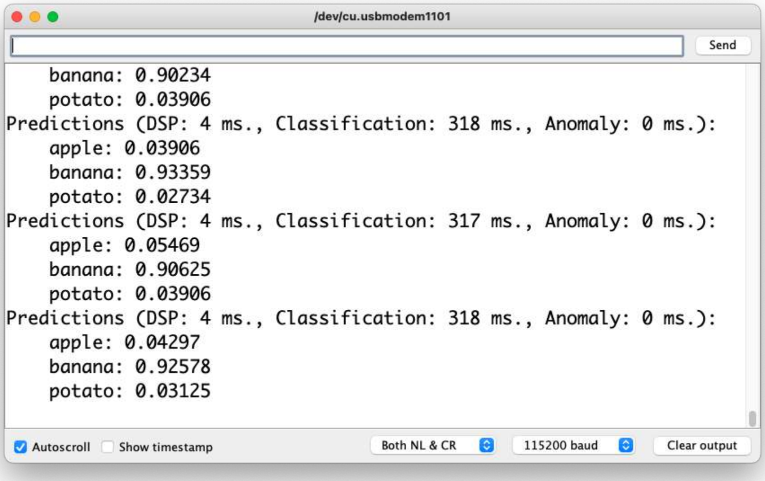
The modified sketch can be downloaded from GitHub: xiao_esp32s3_camera.

> Note that you can optionally keep the pins as a .h file as we did in the Setup Lab.

Upload the code to your XIAO ESP32S3 Sense, and you should be OK to start classifying your fruits and vegetables! You can check the result on Serial Monitor.

Testing the Model (Inference)

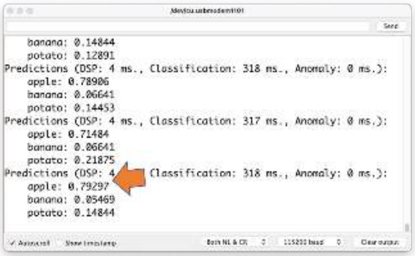
Getting a photo with the camera, the classification result will appear on the Serial Monitor:



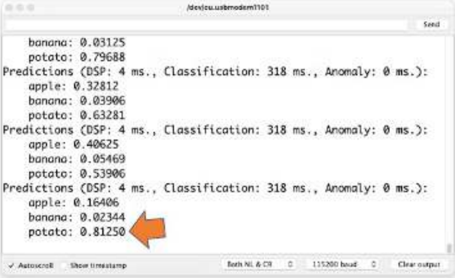
```
    banana: 0.90234
    potato: 0.03906
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
    apple: 0.03906
    banana: 0.93359
    potato: 0.02734
Predictions (DSP: 4 ms., Classification: 317 ms., Anomaly: 0 ms.):
    apple: 0.05469
    banana: 0.90625
    potato: 0.03906
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
    apple: 0.04297
    banana: 0.92578
    potato: 0.03125
```

☑ Autoscroll □ Show timestamp Both NL & CR 115200 baud

Other tests:



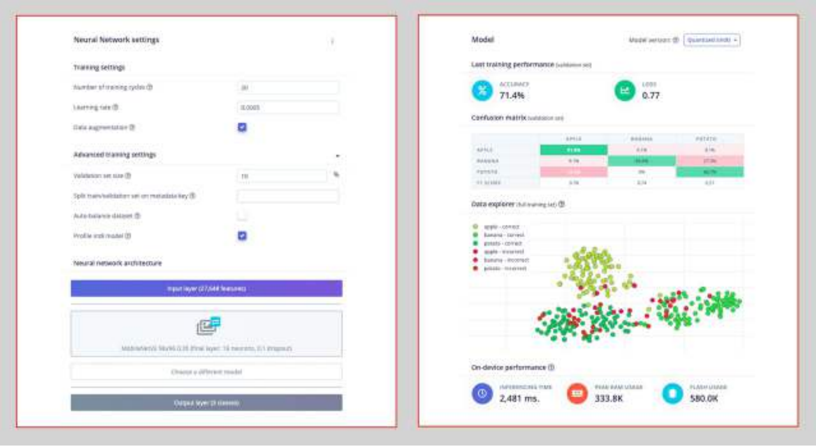
```
    banana: 0.14844
    potato: 0.12891
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
    apple: 0.78906
    banana: 0.06641
    potato: 0.14453
Predictions (DSP: 4 ms., Classification: 317 ms., Anomaly: 0 ms.):
    apple: 0.71484
    banana: 0.06641
    potato: 0.21875
Predictions (DSP: 4     Classification: 318 ms., Anomaly: 0 ms.):
    apple: 0.79297
    banana: 0.05469
    potato: 0.14844
```



```
    banana: 0.03125
    potato: 0.79688
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
    apple: 0.32812
    banana: 0.03906
    potato: 0.63281
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
    apple: 0.40625
    banana: 0.05469
    potato: 0.53906
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
    apple: 0.16406
    banana: 0.02344
    potato: 0.81250
```

Testing with a Bigger Model

Now, let's go to the other side of the model size. Let's select a MobilinetV2 96 × 96 0.35, having as input RGB images.



Neural Network settings

Training settings

Data augmentation ☑

Advanced training settings

Auto-balance dataset □

Profile int8 model ☑

Neural network architecture

Model

Last training performance (validation set)

ACCURACY 71.4%

LOSS 0.77

Confusion matrix (validation set)

Data explorer (full training set)

On-device performance

INFERENCING TIME 2,481 ms.

PEAK RAM USAGE 333.8K

FLASH USAGE 580.0K

Even with a bigger model, the accuracy could be better, and the amount of memory necessary to run the model increases five times, with latency increasing seven times.

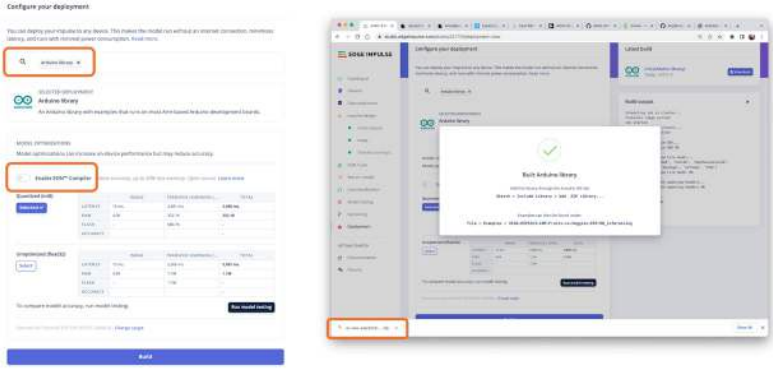
> Note that the performance here is estimated with a smaller device, the ESP-EYE. The actual inference with the ESP32S3 should be better.

To improve our model, we will need to train more images.

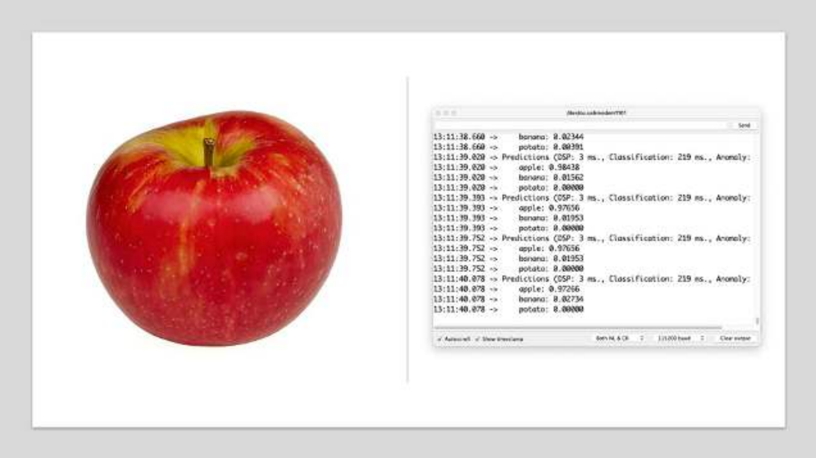
Even though our model did not improve accuracy, let's test whether the XIAO can handle such a bigger model. We will do a simple inference test with the Static Buffer sketch.

Let's redeploy the model. If the EON Compiler is enabled when you generate the library, the total memory needed for inference should be reduced, but it does not influence accuracy.

> **Attention** - The Xiao ESP32S3 with PSRAM enable has enough memory to run the inference, even in such bigger model. Keep the EON Compiler **NOT ENABLED**.

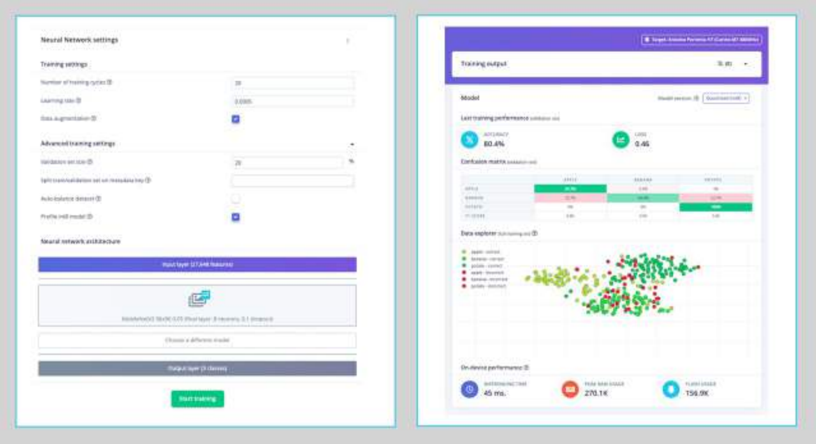


Doing an inference with MobilinetV2 96 × 96 0.35, having as input RGB images, the latency was 219 ms, which is great for such a bigger model.



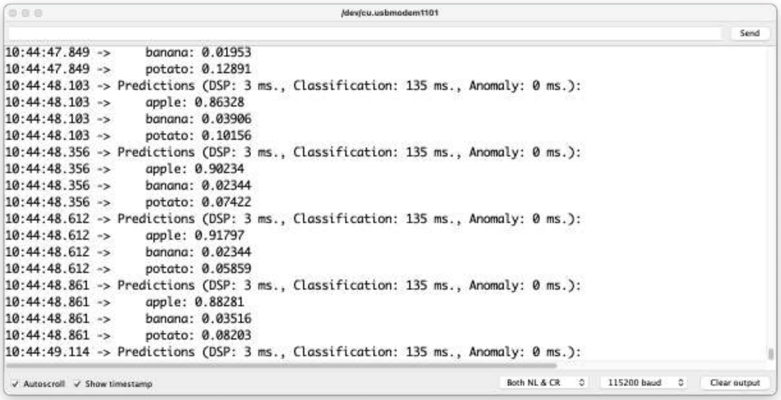
```
13:11:38.660 ->     banana: 0.02344
13:11:38.660 ->     potato: 0.00391
13:11:39.020 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:11:39.020 ->     apple: 0.98438
13:11:39.020 ->     banana: 0.01562
13:11:39.020 ->     potato: 0.00000
13:11:39.393 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:11:39.393 ->     apple: 0.97656
13:11:39.393 ->     banana: 0.01953
13:11:39.393 ->     potato: 0.00000
13:11:39.752 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:11:39.752 ->     apple: 0.97656
13:11:39.752 ->     banana: 0.01953
13:11:39.752 ->     potato: 0.00000
13:11:40.078 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:11:40.078 ->     apple: 0.97266
13:11:40.078 ->     banana: 0.02734
13:11:40.078 ->     potato: 0.00000
```

For the test, we can train the model again, using the smallest version of MobileNet V2, with an alpha of 0.05. Interesting that the result in accuracy was higher.



> Note that the estimated latency for an Arduino Portenta (or Nicla), running with a clock of 480 MHz is 45 ms.

Deploying the model, we got an inference of only 135 ms, remembering that the XIAO runs with half of the clock used by the Portenta/Nicla (240 MHz):



```
10:44:47.849 ->    banana: 0.01953
10:44:47.849 ->    potato: 0.12891
10:44:48.103 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
10:44:48.103 ->    apple: 0.86328
10:44:48.103 ->    banana: 0.03906
10:44:48.103 ->    potato: 0.10156
10:44:48.356 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
10:44:48.356 ->    apple: 0.90234
10:44:48.356 ->    banana: 0.02344
10:44:48.356 ->    potato: 0.07422
10:44:48.612 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
10:44:48.612 ->    apple: 0.91797
10:44:48.612 ->    banana: 0.02344
10:44:48.612 ->    potato: 0.05859
10:44:48.861 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
10:44:48.861 ->    apple: 0.88281
10:44:48.861 ->    banana: 0.03516
10:44:48.861 ->    potato: 0.08203
10:44:49.114 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
```

Running inference on the SenseCraft-Web-Toolkit

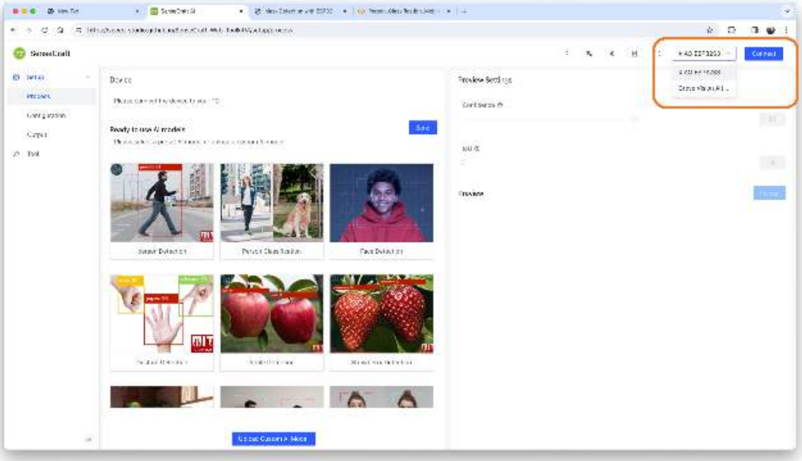
One significant limitation of viewing inference on Arduino IDE is that we can not see what the camera focuses on. A good alternative is the **SenseCraft-Web-Toolkit**, a visual model deployment tool provided by SSCMA(Seeed SenseCraft Model Assistant). This tool allows you to deploy models to various platforms easily through simple operations. The tool offers a user-friendly interface and does not require any coding.

Follow the following steps to start the SenseCraft-Web-Toolkit:

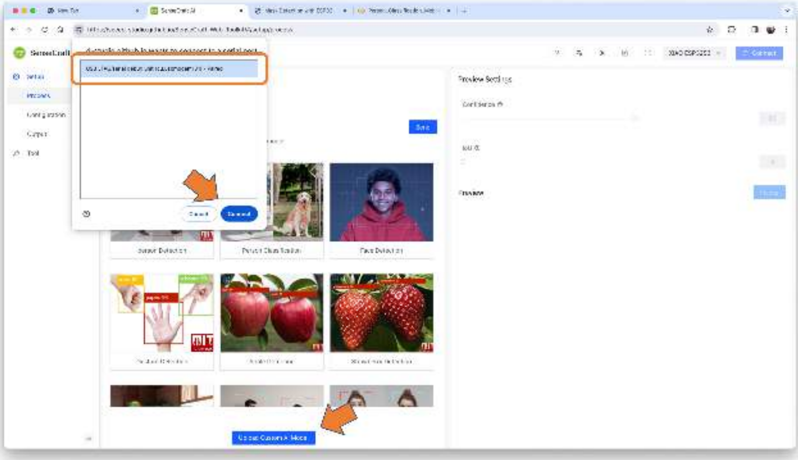
1. Open the SenseCraft-Web-Toolkit website.

2. Connect the XIAO to your computer:

- Having the XIAO connected, select it as below:



- Select the device/Port and press [Connect]:



You can try several Computer Vision models previously uploaded by Seeed Studio. Try them and have fun!

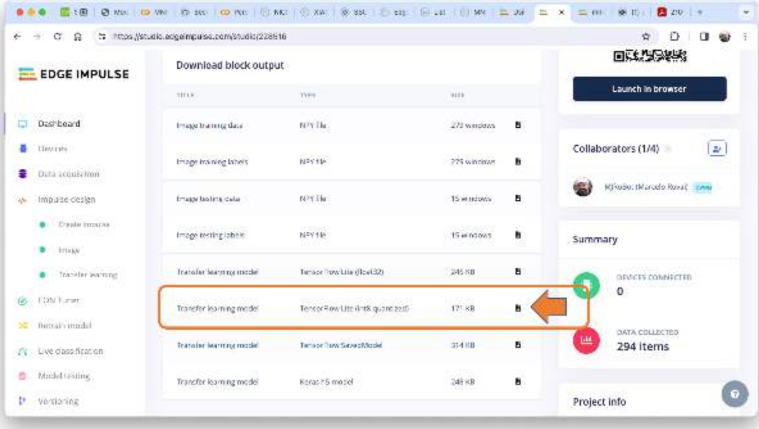
In our case, we will use the blue button at the bottom of the page: [Upload Custom AI Model].

But first, we must download from Edge Impulse Studio our **quantized.tflite** model.

3. Go to your project at Edge Impulse Studio, or clone this one:

- XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v1-ESP-NN

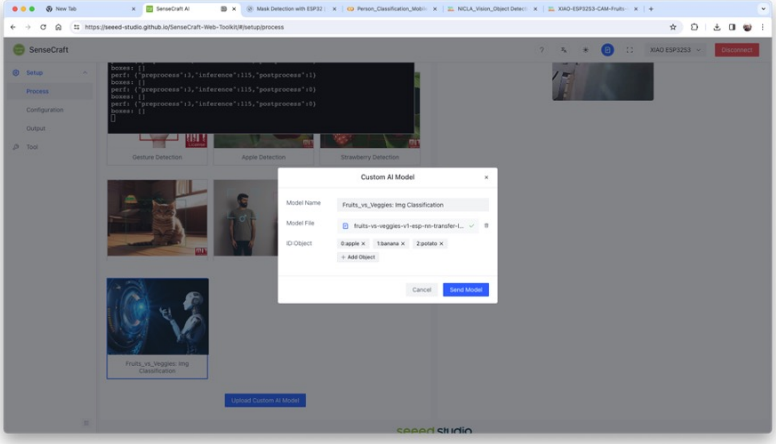
4. On the `Dashboard`, download the model ("block output"): `Transfer learning model - TensorFlow Lite (int8 quantized)`.



Download block output

TITLE	TYPE	SIZE
Image training data	NPY file	273 windows
Image training labels	NPY file	273 windows
Image testing data	NPY file	15 windows
Image testing labels	NPY file	15 windows
Transfer learning model	TensorFlow Lite (float32)	245 KB
Transfer learning model	TensorFlow Lite (int8 quantized)	17[illegible] KB
Transfer learning model	TensorFlow SavedModel	314 KB
Transfer learning model	Keras h5 model	248 KB

5. On SenseCraft-Web-Toolkit, use the blue button at the bottom of the page: `[Upload Custom AI Model]`. A window will pop up. Enter the Model file that you downloaded to your computer from Edge Impulse Studio, choose a Model Name, and enter with labels (ID: Object):



```
boxes: []
perf: {"preprocess":3,"inference":115,"postprocess":1}
boxes: []
perf: {"preprocess":3,"inference":115,"postprocess":0}
boxes: []
perf: {"preprocess":3,"inference":115,"postprocess":0}
boxes: []
```

Custom AI Model

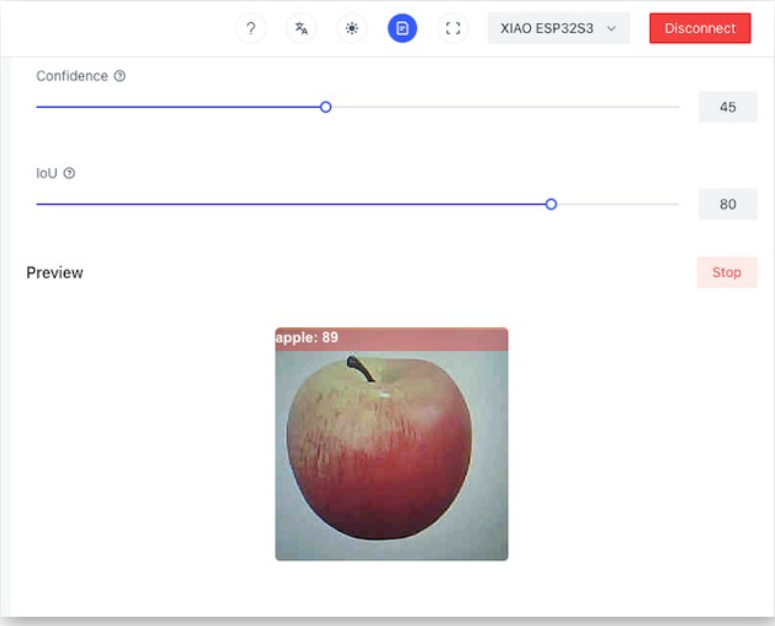
Model Name: Fruits_vs_Veggies: Img Classification

Model File: fruits-vs-veggies-v1-esp-nn-transfer-l...

ID:Object: 0:apple 1:banana 2:potato

Note that you should use the labels trained on EI Studio, entering them in alphabetic order (in our case: apple, banana, potato).

After a few seconds (or minutes), the model will be uploaded to your device, and the camera image will appear in real-time on the Preview Sector:

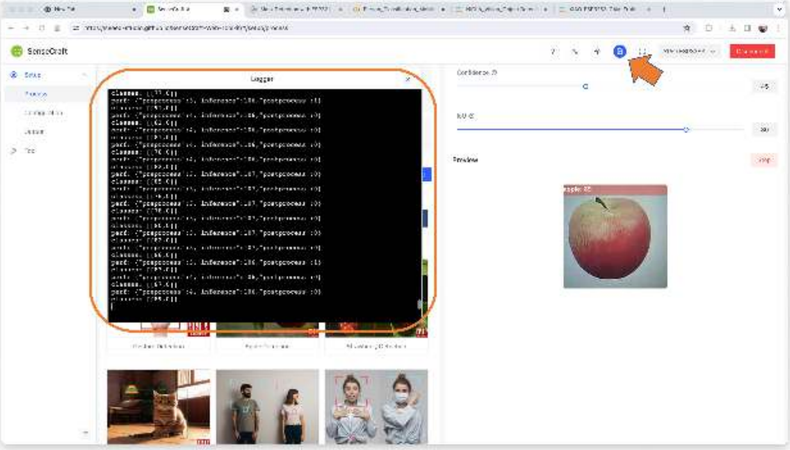


Confidence 45

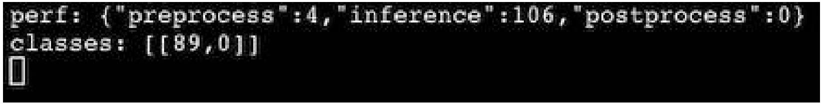
IoU 80

The Classification result will be at the top of the image. You can also select the Confidence of your inference cursor `Confidence`.

Clicking on the top button (Device Log), you can open a Serial Monitor to follow the inference, the same that we have done with the Arduino IDE:



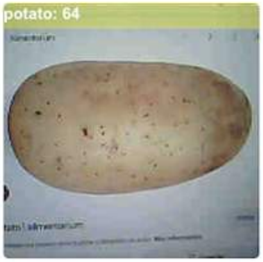
On Device Log, you will get information as:

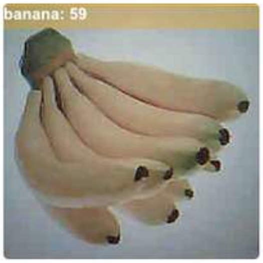


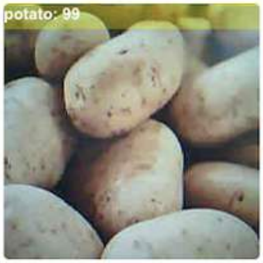
```
perf: {"preprocess":4,"inference":106,"postprocess":0}
classes: [[89,0]]
```

- Preprocess time (image capture and Crop): 4 ms,
- Inference time (model latency): 106 ms,
- Postprocess time (display of the image and inclusion of data): 0 ms,
- Output tensor (classes), for example: [[89,0]]; where 0 is Apple (and 1is banana and 2 is potato).

Here are other screenshots:







Conclusion

The XIAO ESP32S3 Sense is very flexible, inexpensive, and easy to program. The project proves the potential of TinyML. Memory is not an issue; the device can handle many post-processing tasks, including communication.

You will find the last version of the codeon the GitHub repository: XIAO-ESP32S3-Sense.

Resources

- XIAO ESP32S3 Codes
- Dataset
- Edge Impulse Project

Object Detection

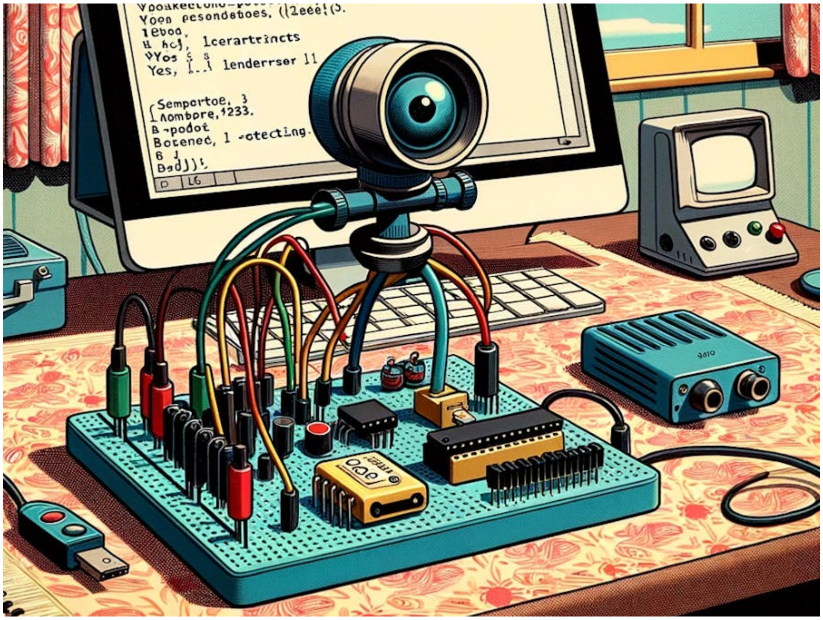


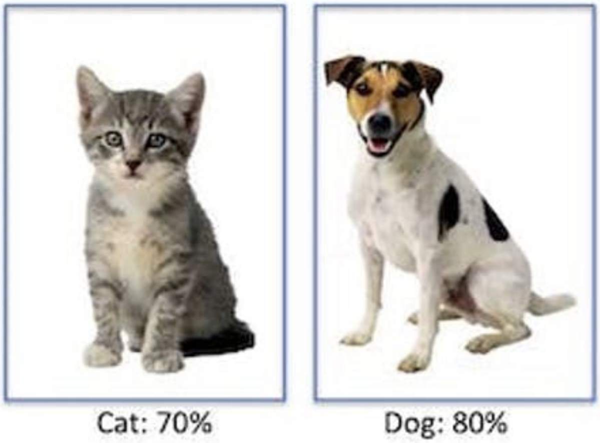
Figure 20.11: *DALL·E prompt - Cartoon styled after 1950s animations, showing a detailed board with sensors, particularly a camera, on a table with patterned cloth. Behind the board, a computer with a large back showcases the Arduino IDE. The IDE's content hints at LED pin assignments and machine learning inference for detecting spoken commands. The Serial Monitor, in a distinct window, reveals outputs for the commands 'yes' and 'no'.*

Overview

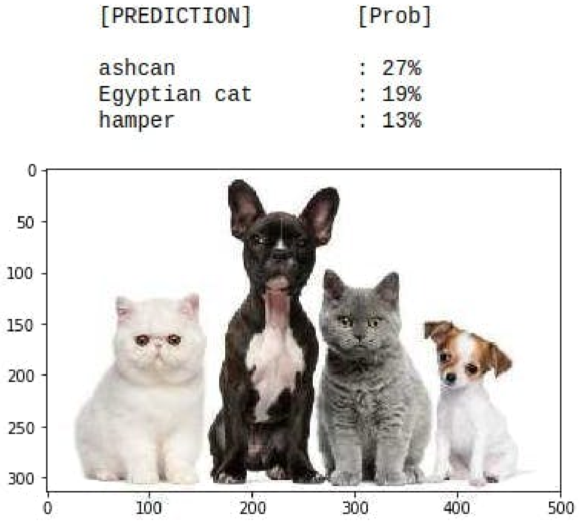
In the last section regarding Computer Vision (CV) and the XIAO ESP32S3, *Image Classification,* we learned how to set up and classify images with this remarkable development board. Continuing our CV journey, we will explore **Object Detection** on microcontrollers.

Object Detection versus Image Classification

The main task with Image Classification models is to identify the most probable object category present on an image, for example, to classify between a cat or a dog, dominant "objects" in an image:



But what happens if there is no dominant category in the image?



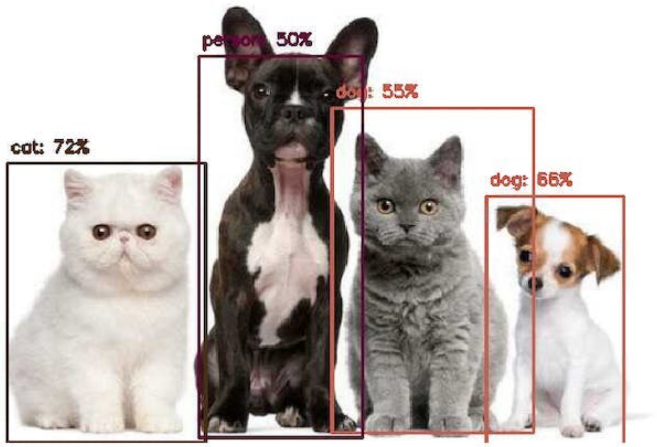
```
[PREDICTION]          [Prob]

ashcan                : 27%
Egyptian cat          : 19%
hamper                : 13%
```

An image classification model identifies the above image utterly wrong as an "ashcan," possibly due to the color tonalities.

> The model used in the previous images is MobileNet, which is trained with a large dataset, *ImageNet*, running on a Raspberry Pi.

To solve this issue, we need another type of model, where not only **multiple categories** (or labels) can be found but also **where** the objects are located on a given image.

As we can imagine, such models are much more complicated and bigger, for example, the **MobileNetV2 SSD FPN-Lite 320x320, trained with the COCO dataset.** This pre-trained object detection model is designed to locate up to 10 objects within an image, outputting a bounding box for each object detected. The below image is the result of such a model running on a Raspberry Pi:



Those models used for object detection (such as the MobileNet SSD or YOLO) usually have several MB in size, which is OK for use with Raspberry Pi but unsuitable for use with embedded devices, where the RAM usually has, at most, a few MB as in the case of the XIAO ESP32S3.

An Innovative Solution for Object Detection: FOMO

Edge Impulse launched in 2022, **FOMO** (Faster Objects, More Objects), a novel solution to perform object detection on embedded devices, such as the Nicla Vision and Portenta (Cortex M7), on Cortex M4F CPUs (Arduino Nano33 and OpenMV M4 series) as well the Espressif ESP32 devices (ESP-CAM, ESP-EYE and XIAO ESP32S3 Sense).

In this Hands-On project, we will explore Object Detection using FOMO.

> To understand more about FOMO, you can go into the official FOMO announcement by Edge Impulse, where Louis Moreau and Mat Kelcey explain in detail how it works.

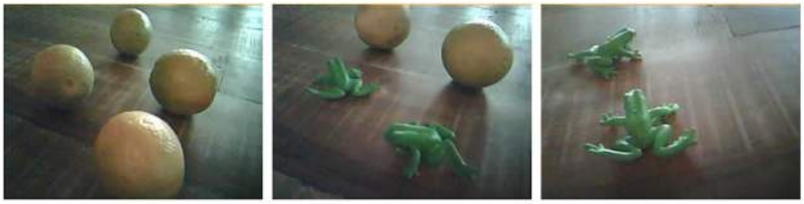
The Object Detection Project Goal

All Machine Learning projects need to start with a detailed goal. Let’s assume we are in an industrial or rural facility and must sort and count **oranges (fruits)** and particular **frogs (bugs)**.

In other words, we should perform a multi-label classification, where each image can have three classes:

- Background (No objects)
- Fruit
- Bug

Here are some not labeled image samples that we should use to detect the objects (fruits and bugs):



We are interested in which object is in the image, its location (centroid), and how many we can find on it. The object's size is not detected with FOMO, as with MobileNet SSD or YOLO, where the Bounding Box is one of the model outputs.

We will develop the project using the XIAO ESP32S3 for image capture and model inference. The ML project will be developed using the Edge Impulse Studio. But before starting the object detection project in the Studio, let's create a *raw dataset* (not labeled) with images that contain the objects to be detected.

Data Collection

You can capture images using the XIAO, your phone, or other devices. Here, we will use the XIAO with code from the Arduino IDE ESP32 library.

Collecting Dataset with the XIAO ESP32S3

Open the Arduino IDE and select the XIAO_ESP32S3 board (and the port where it is connected). On `File > Examples > ESP32 > Camera`, select `CameraWebServer`.

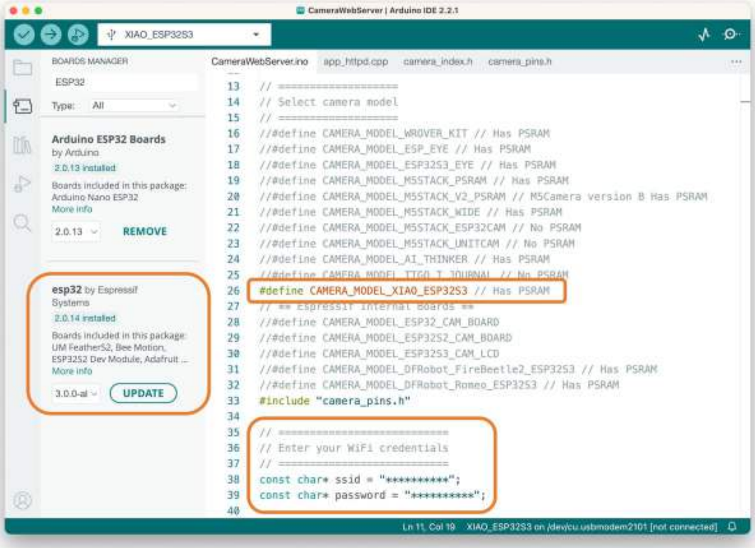
On the BOARDS MANAGER panel, confirm that you have installed the latest "stable" package.

> **Attention**
>
> Alpha versions (for example, 3.x-alpha) do not work correctly with the XIAO and Edge Impulse. Use the last stable version (for example, 2.0.11) instead.

You also should comment on all cameras' models, except the XIAO model pins:

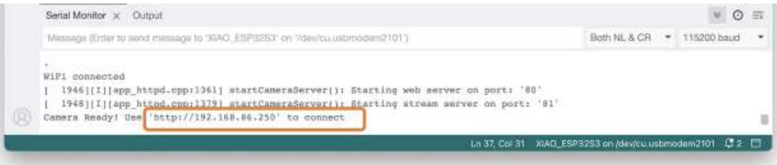
`#define CAMERA_MODEL_XIAO_ESP32S3 // Has PSRAM`

And on `Tools`, enable the PSRAM. Enter your wifi credentials and upload the code to the device:


CameraWebServer | Arduino IDE 2.2.1
XIAO_ESP32S3
BOARDS MANAGER
ESP32
Type: All
Arduino ESP32 Boards
by Arduino
2.0.13 installed
Boards included in this package: Arduino Nano ESP32
More info
2.0.13
REMOVE
esp32 by Espressif Systems
2.0.14 installed
Boards included in this package: UM FeatherS2, Bee Motion, ESP32S2 Dev Module, Adafruit ...
More info
3.0.0-al
UPDATE
CameraWebServer.ino
app_httpd.cpp
camera_index.h
camera_pins.h
Ln 11, Col 19 XIAO_ESP32S3 on /dev/cu.usbmodem2101 [not connected]

```
// ===================
// Select camera model
// ===================
//#define CAMERA_MODEL_WROVER_KIT // Has PSRAM
//#define CAMERA_MODEL_ESP_EYE // Has PSRAM
//#define CAMERA_MODEL_ESP32S3_EYE // Has PSRAM
//#define CAMERA_MODEL_M5STACK_PSRAM // Has PSRAM
//#define CAMERA_MODEL_M5STACK_V2_PSRAM // M5Camera version B Has PSRAM
//#define CAMERA_MODEL_M5STACK_WIDE // Has PSRAM
//#define CAMERA_MODEL_M5STACK_ESP32CAM // No PSRAM
//#define CAMERA_MODEL_M5STACK_UNITCAM // No PSRAM
//#define CAMERA_MODEL_AI_THINKER // Has PSRAM
//#define CAMERA_MODEL_TTGO_T_JOURNAL // No PSRAM
#define CAMERA_MODEL_XIAO_ESP32S3 // Has PSRAM
// ** Espressif Internal Boards **
//#define CAMERA_MODEL_ESP32_CAM_BOARD
//#define CAMERA_MODEL_ESP32S2_CAM_BOARD
//#define CAMERA_MODEL_ESP32S3_CAM_LCD
//#define CAMERA_MODEL_DFRobot_FireBeetle2_ESP32S3 // Has PSRAM
//#define CAMERA_MODEL_DFRobot_Romeo_ESP32S3 // Has PSRAM
#include "camera_pins.h"

// ===========================
// Enter your WiFi credentials
// ===========================
const char* ssid = "**********";
const char* password = "**********";

```

If the code is executed correctly, you should see the address on the Serial Monitor:

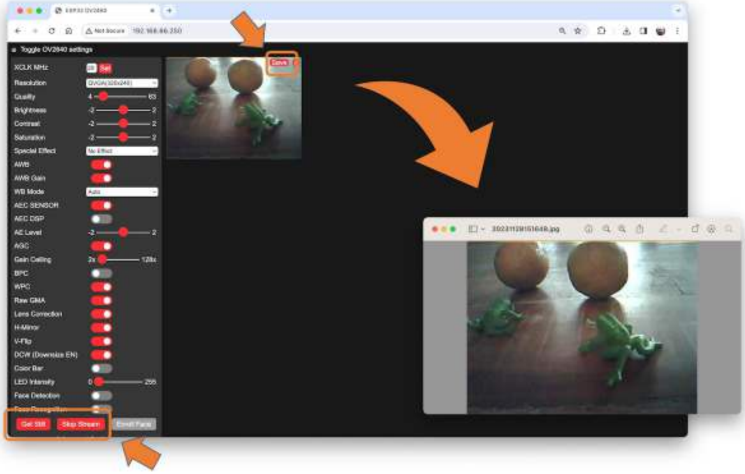


Serial Monitor
Output
Message (Enter to send message to 'XIAO_ESP32S3' on '/dev/cu.usbmodem2101')
Both NL & CR
115200 baud

```
.
WiFi connected
[  1946][I][app_httpd.cpp:1361] startCameraServer(): Starting web server on port: '80'
[  1948][I][app_httpd.cpp:1379] startCameraServer(): Starting stream server on port: '81'
Camera Ready! Use 'http://192.168.86.250' to connect
```

Ln 37, Col 31 XIAO_ESP32S3 on /dev/cu.usbmodem2101

Copy the address on your browser and wait for the page to be uploaded. Select the camera resolution (for example, QVGA) and select `[START STREAM]`. Wait for a few seconds/minutes, depending on your connection. You can save an image on your computer download area using the [Save] button.



Edge impulse suggests that the objects should be similar in size and not overlapping for better performance. This is OK in an industrial facility, where the camera should be fixed, keeping the same distance from the objects to be detected. Despite that, we will also try using mixed sizes and positions to see the result.

> We do not need to create separate folders for our images because each contains multiple labels.

We suggest using around 50 images to mix the objects and vary the number of each appearing on the scene. Try to capture different angles, backgrounds, and light conditions.

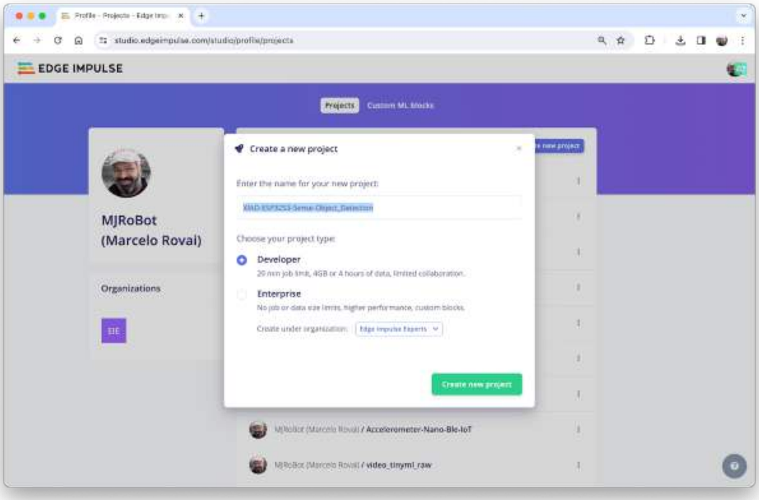
> The stored images use a QVGA frame size of 320 × 240 and RGB565 (color pixel format).

After capturing your dataset, `[Stop Stream]` and move your images to a folder.

Edge Impulse Studio

Setup the project

Go to Edge Impulse Studio, enter your credentials at **Login** (or create an account), and start a new project.



studio.edgeimpulse.com/studio/profile/projects

Create a new project

Enter the name for your new project: XIAO-ESP32S3-Sense-Object_Detection

Choose your project type:

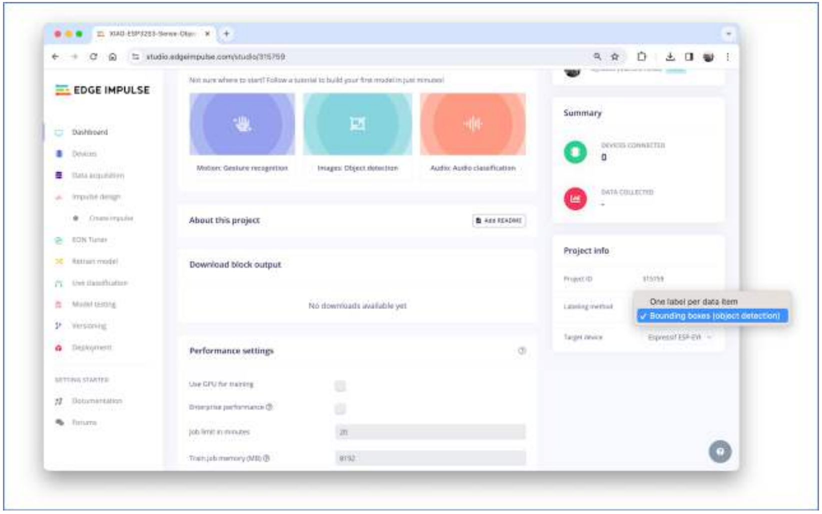
● Developer
20 min job limit, 4GB or 4 hours of data, limited collaboration.

○ Enterprise
No job or data size limits, higher performance, custom blocks.
Create under organization: Edge Impulse Experts

Profile - Projects - Edge Imp
EDGE IMPULSE
Projects
Custom ML blocks
MJRoBot
(Marcelo Rovai)
Organizations
EE
Create new project
MJRoBot (Marcelo Rovai) / Accelerometer-Nano-Ble-IoT
MJRoBot (Marcelo Rovai) / video_tinyml_raw

Here, you can clone the project developed for this hands-on: XIAO-ESP32S3-Sense-Object_Detection

On your Project Dashboard, go down and on **Project info** and select **Bounding boxes (object detection)** and **Espressif ESP-EYE** (most similar to our board) as your Target Device:



studio.edgeimpulse.com/studio/315759

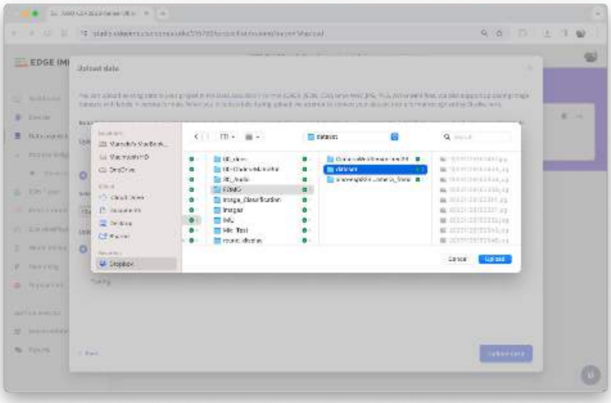
Not sure where to start? Follow a tutorial to build your first model in just minutes!

Project info
Project ID: 315759
Labeling method: One label per data item / ✓ Bounding boxes (object detection)
Target device: Espressif ESP-EYE

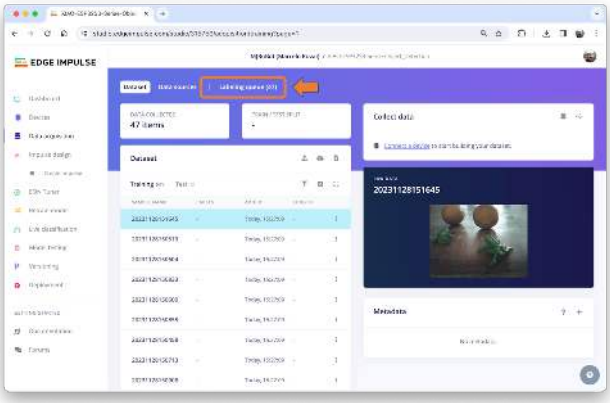
Performance settings
Use GPU for training □
Enterprise performance □
Job limit in minutes: 20
Train job memory (MB): 8192

Uploading the unlabeled data

On Studio, go to the `Data acquisition` tab, and on the `UPLOAD DATA` section, upload files captured as a folder from your computer.



You can leave for the Studio to split your data automatically between Train and Test or do it manually. We will upload all of them as training.



All the not-labeled images (47) were uploaded but must be labeled appropriately before being used as a project dataset. The Studio has a tool for that purpose, which you can find in the link Labeling queue (47).

There are two ways you can use to perform AI-assisted labeling on the Edge Impulse Studio (free version):

- Using yolov5
- Tracking objects between frames

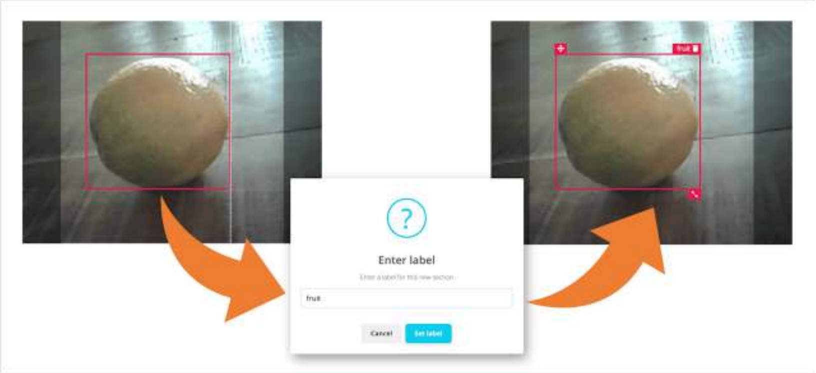
> Edge Impulse launched an auto-labeling feature for Enterprise customers, easing labeling tasks in object detection projects.

Ordinary objects can quickly be identified and labeled using an existing library of pre-trained object detection models from YOLOv5 (trained with the COCO dataset). But since, in our case, the objects are not part of COCO datasets, we should select the option of tracking objects. With this option, once you draw bounding boxes and label the images in one frame, the objects will be tracked automatically from frame to frame, *partially* labeling the new ones (not all are correctly labeled).

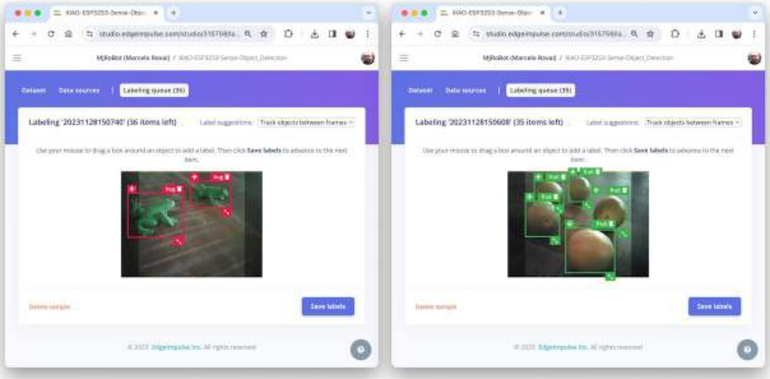
> You can use the EI uploader to import your data if you already have a labeled dataset containing bounding boxes.

Labeling the Dataset

Starting with the first image of your unlabeled data, use your mouse to drag a box around an object to add a label. Then click **Save labels** to advance to the next item.



Continue with this process until the queue is empty. At the end, all images should have the objects labeled as those samples below:


Labeling '20231128150740' (36 items left)

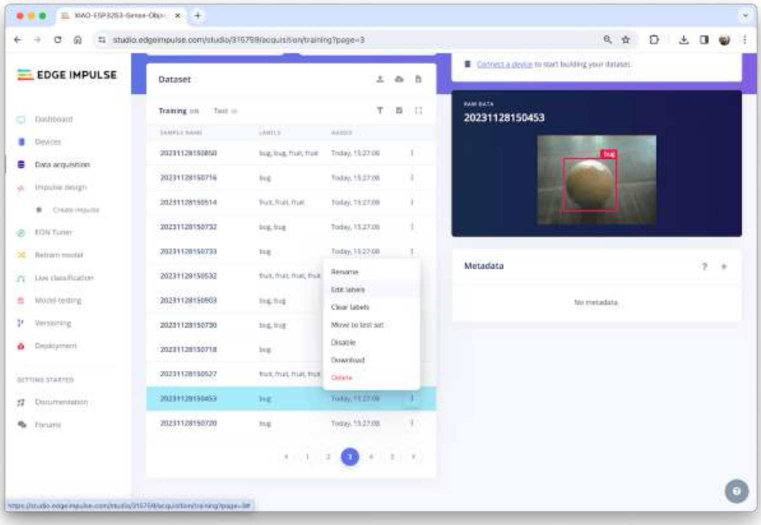
Use your mouse to drag a box around an object to add a label. Then click **Save labels** to advance to the next item.

Labeling '20231128150608' (35 items left)

Use your mouse to drag a box around an object to add a label. Then click **Save labels** to advance to the next item.

studio.edgeimpulse.com/studio/315768...
MjRoBot (Marcelo Rovai) / XIAO-ESP32S3-Sense-Object_Detection
Dataset
Data sources
Labeling queue (36)
Labeling queue (35)
Label suggestions: Track objects between frames
frog
fruit
Delete sample
Save labels
© 2023 EdgeImpulse Inc. All rights reserved

Next, review the labeled samples on the `Data acquisition` tab. If one of the labels is wrong, you can edit it using the *three dots* menu after the sample name:



Dataset

Training Test

SAMPLE NAME	LABELS	ADDED
20231128150850	bug, bug, fruit, fruit	Today, 15:27:08
20231128150716	bug	Today, 15:27:08
20231128150514	fruit, fruit, fruit	Today, 15:27:08
20231128150732	bug, bug	Today, 15:27:08
20231128150733	bug	Today, 15:27:08
20231128150532	fruit, fruit, fruit, fruit	
20231128150903	bug, bug	
20231128150730	bug, bug	
20231128150718	bug	
20231128150527	fruit, fruit, fruit, fruit	
20231128150453	bug	Today, 15:27:08
20231128150720	bug	Today, 15:27:08

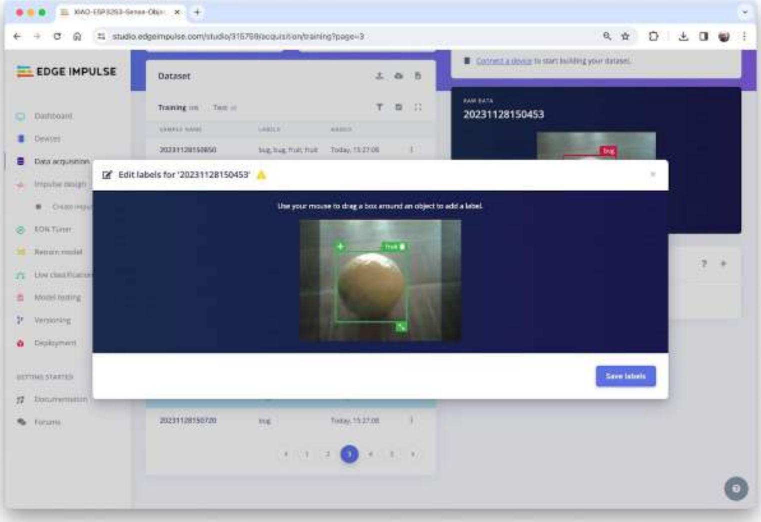
Connect a device to start building your dataset.

RAW DATA
20231128150453

Metadata

No metadata

You will be guided to replace the wrong label and correct the dataset.

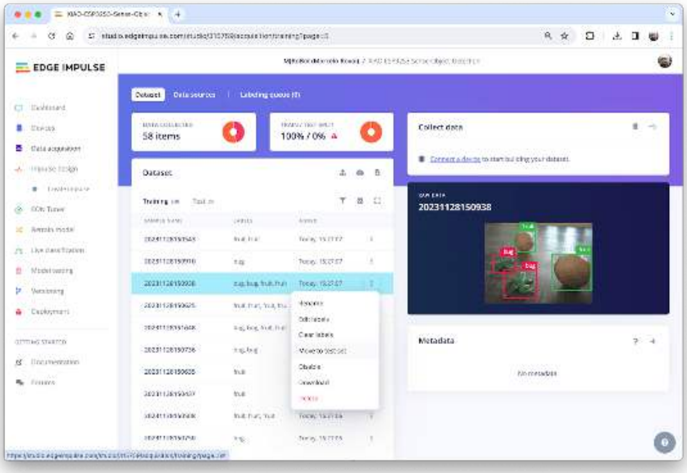


Edit labels for '20231128150453'

Use your mouse to drag a box around an object to add a label.

Balancing the dataset and split Train/Test

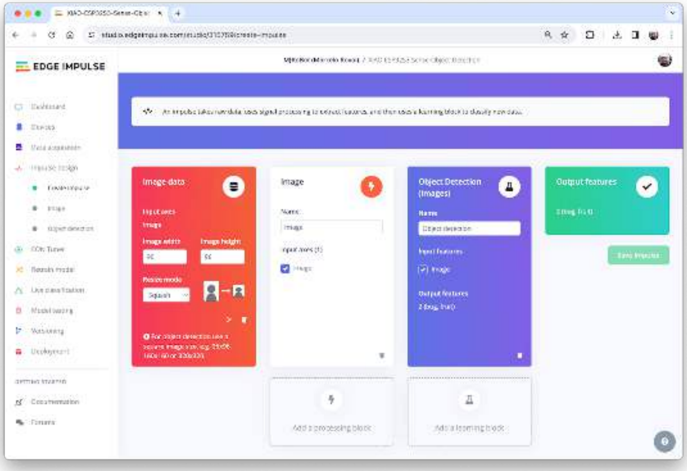
After labeling all data, it was realized that the class fruit had many more samples than the bug. So, 11 new and additional bug images were collected (ending with 58 images). After labeling them, it is time to select some images and move them to the test dataset. You can do it using the three-dot menu after the image name. I selected six images, representing 13% of the total dataset.



The Impulse Design

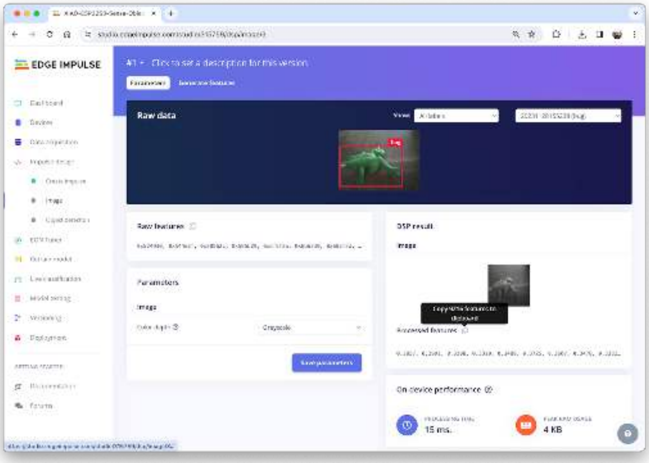
In this phase, you should define how to:

- **Pre-processing** consists of resizing the individual images from 320 × 240 to 96 × 96 and squashing them (squared form, without cropping). Afterward, the images are converted from RGB to Grayscale.
- **Design a Model,** in this case, "Object Detection."



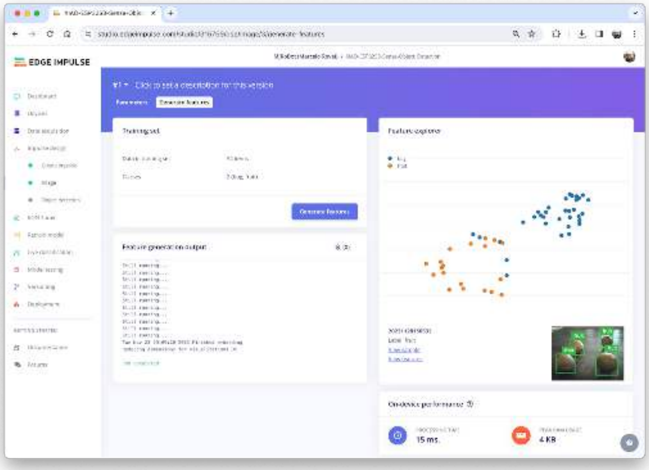
Preprocessing all dataset

In this section, select **Color depth** as Grayscale, suitable for use with FOMO models and Save parameters.



Parameters
Image
Color depth: Grayscale

The Studio moves automatically to the next section, Generate features, where all samples will be pre-processed, resulting in a dataset with individual 96 × 96 × 1 images or 9,216 features.



The feature explorer shows that all samples evidence a good separation after the feature generation.

> Some samples seem to be in the wrong space, but clicking on them confirms the correct labeling.

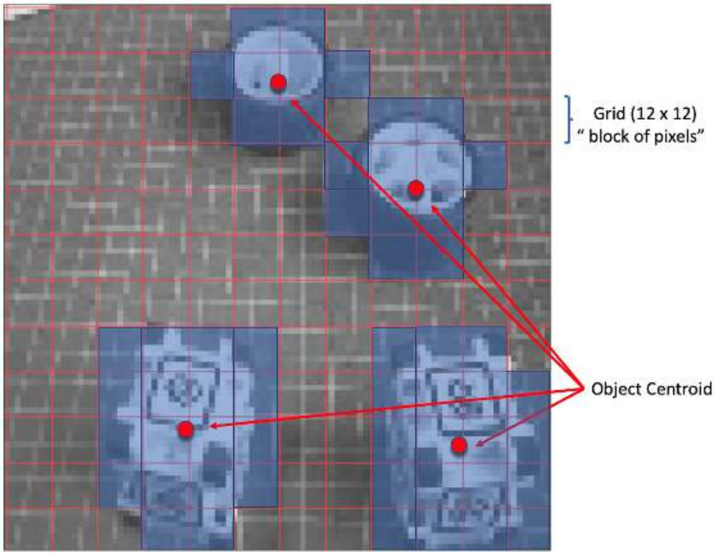
Model Design, Training, and Test

We will use FOMO, an object detection model based on MobileNetV2 (alpha 0.35) designed to coarsely segment an image into a grid of **background** vs **objects of interest** (here, *boxes* and *wheels*).

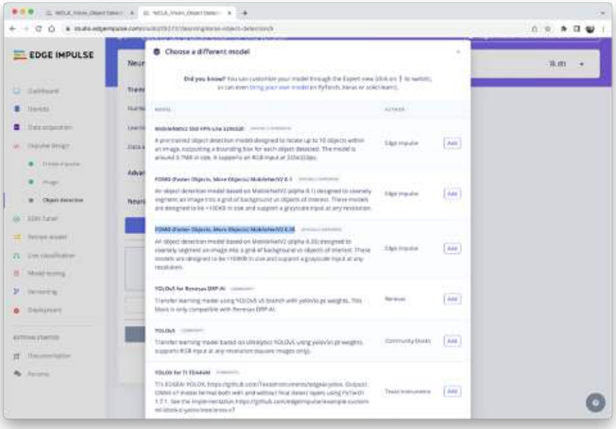
FOMO is an innovative machine learning model for object detection, which can use up to 30 times less energy and memory than traditional models like Mobilenet SSD and YOLOv5. FOMO can operate on microcontrollers with less than 200 KB of RAM. The main reason this is possible is that while other models calculate the object's size by drawing a square around it (bounding box), FOMO ignores the size of the image, providing only the information about where the object is located in the image through its centroid coordinates.

How FOMO works?

FOMO takes the image in grayscale and divides it into blocks of pixels using a factor of 8. For the input of 96×96, the grid would be 12×12 $(96/8 = 12)$. Next, FOMO will run a classifier through each pixel block to calculate the probability that there is a box or a wheel in each of them and, subsequently, determine the regions that have the highest probability of containing the object (If a pixel block has no objects, it will be classified as *background*). From the overlap of the final region, the FOMO provides the coordinates (related to the image dimensions) of the centroid of this region.



For training, we should select a pre-trained model. Let's use the **FOMO (Faster Objects, More Objects) MobileNetV2 0.35.** This model uses around 250 KB of RAM and 80 KB of ROM (Flash), which suits well with our board.



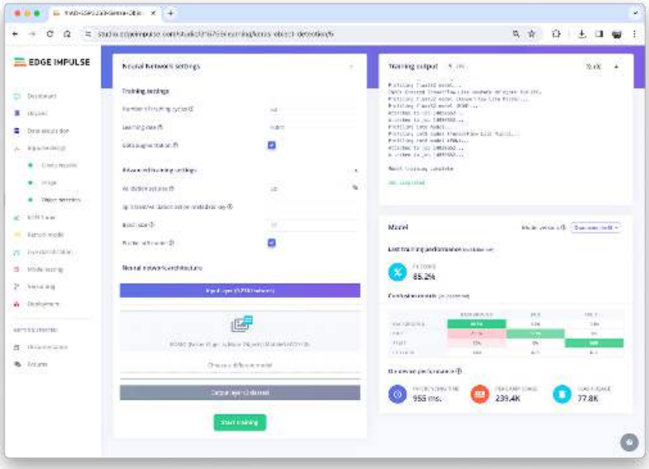
Regarding the training hyper-parameters, the model will be trained with:

- Epochs: 60
- Batch size: 32
- Learning Rate: 0.001.

For validation during training, 20% of the dataset (*validation_dataset*) will be spared. For the remaining 80% (*train_dataset*), we will apply Data Augmentation, which will randomly flip, change the size and brightness of the image, and crop them, artificially increasing the number of samples on the dataset for training.

As a result, the model ends with an overall F1 score of 85%, similar to the result when using the test data (83%).

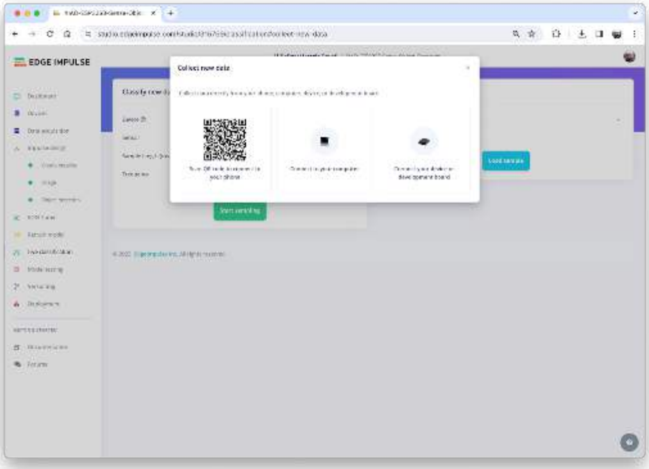
> Note that FOMO automatically added a 3rd label background to the two previously defined (*box* and *wheel*).



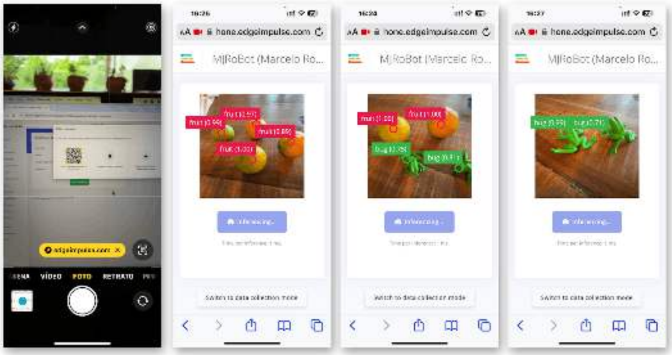
In object detection tasks, accuracy is generally not the primary evaluation metric. Object detection involves classifying objects and providing bounding boxes around them, making it a more complex problem than simple classification. The issue is that we do not have the bounding box, only the centroids. In short, using accuracy as a metric could be misleading and may not provide a complete understanding of how well the model is performing. Because of that, we will use the F1 score.

Test model with "Live Classification"

Once our model is trained, we can test it using the Live Classification tool. On the correspondent section, click on Connect a development board icon (a small MCU) and scan the QR code with your phone.



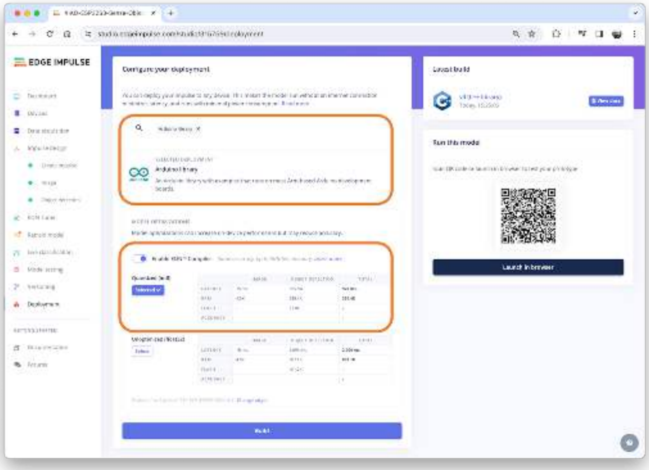
Once connected, you can use the smartphone to capture actual images to be tested by the trained model on Edge Impulse Studio.



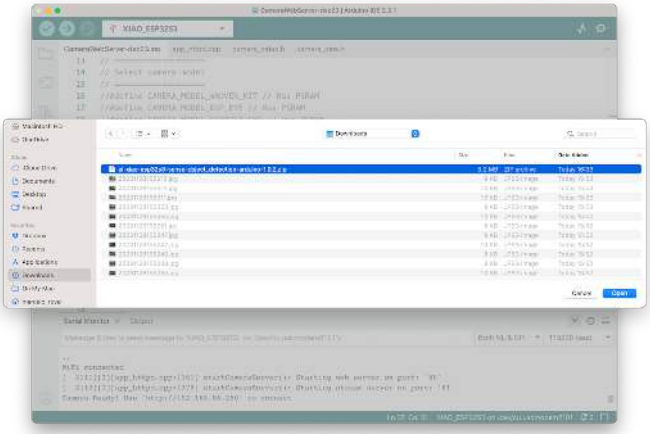
One thing to be noted is that the model can produce false positives and negatives. This can be minimized by defining a proper Confidence Threshold (use the Three dots menu for the setup). Try with 0.8 or more.

Deploying the Model (Arduino IDE)

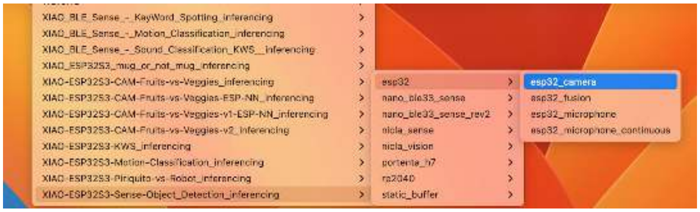
Select the Arduino Library and Quantized (int8) model, enable the EON Compiler on the Deploy Tab, and press [Build].



Open your Arduino IDE, and under Sketch, go to Include Library and add.ZIP Library. Select the file you download from Edge Impulse Studio, and that's it!



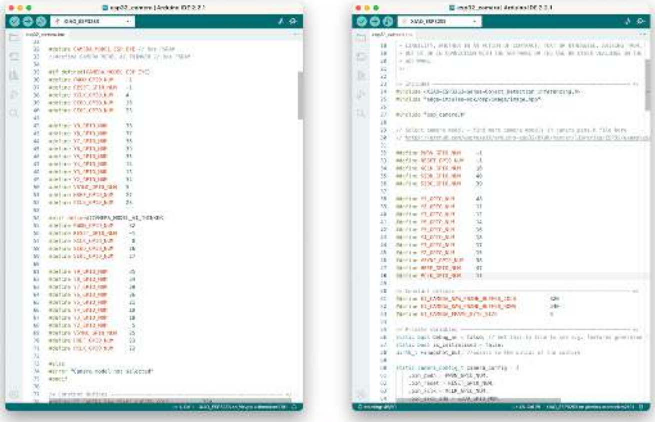
Under the Examples tab on Arduino IDE, you should find a sketch code (`esp32 > esp32_camera`) under your project name.


XIAO_BLE_Sense_-_KeyWord_Spotting_inferencing
XIAO_BLE_Sense_-_Motion_Classification_inferencing
XIAO_BLE_Sense_-_Sound_Classification_KWS__inferencing
XIAO_ESP32S3_mug_or_not_mug_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies-ESP-NN_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v1-ESP-NN_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v2_inferencing
XIAO-ESP32S3-KWS_inferencing
XIAO-ESP32S3-Motion-Classification_inferencing
XIAO-ESP32S3-Pinquito-vs-Robot_inferencing
XIAO-ESP32S3-Sense-Object_Detection_inferencing
esp32
nano_ble33_sense
nano_ble33_sense_rev2
nicla_sense
nicla_vision
portenta_h7
rp2040
static_buffer
esp32_camera
esp32_fusion
esp32_microphone
esp32_microphone_continuous

You should change lines 32 to 75, which define the camera model and pins, using the data related to our model. Copy and paste the below lines, replacing the lines 32-75:

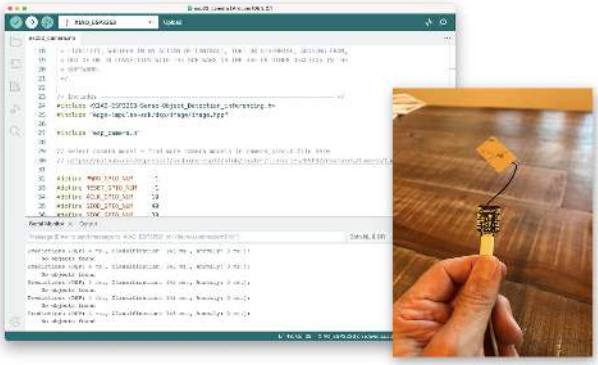
```
#define PWDN_GPIO_NUM     -1
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM     10
#define SIOD_GPIO_NUM     40
#define SIOC_GPIO_NUM     39
#define Y9_GPIO_NUM       48
#define Y8_GPIO_NUM       11
#define Y7_GPIO_NUM       12
#define Y6_GPIO_NUM       14
#define Y5_GPIO_NUM       16
#define Y4_GPIO_NUM       18
#define Y3_GPIO_NUM       17
#define Y2_GPIO_NUM       15
#define VSYNC_GPIO_NUM    38
#define HREF_GPIO_NUM     47
#define PCLK_GPIO_NUM     13
```

Here you can see the resulting code:



Upload the code to your XIAO ESP32S3 Sense, and you should be OK to start detecting fruits and bugs. You can check the result on Serial Monitor.

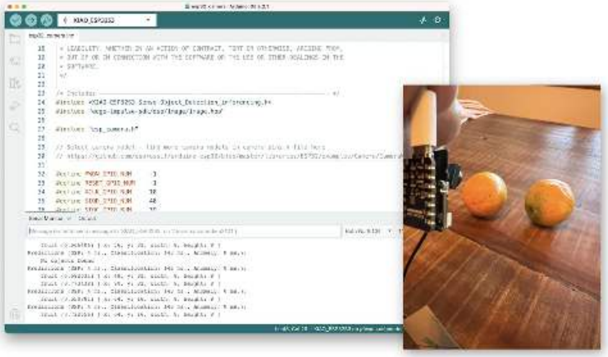
Background



```
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */

/* Includes ---------------------------------------------------------------- */
#include <XIAO-ESP32S3-Sense-Object_Detection_inferencing.h>
#include "edge-impulse-sdk/dsp/image/image.hpp"

#include "esp_camera.h"

// Select camera model - find more camera models in camera_pins.h file here
// https://github.com/espressif/arduino-esp32/blob/master/libraries/ESP32/examples/Camera/C

#define PWDN_GPIO_NUM     -1
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM     10
#define SIOD_GPIO_NUM     40
#define SIOC_GPIO_NUM     39
```

```
Predictions (DSP: 4 ms., Classification: 161 ms., Anomaly: 0 ms.):
    No objects found
Predictions (DSP: 4 ms., Classification: 161 ms., Anomaly: 0 ms.):
    No objects found
Predictions (DSP: 3 ms., Classification: 161 ms., Anomaly: 0 ms.):
    No objects found
Predictions (DSP: 3 ms., Classification: 161 ms., Anomaly: 0 ms.):
    No objects found
Predictions (DSP: 3 ms., Classification: 161 ms., Anomaly: 0 ms.):
    No objects found
```

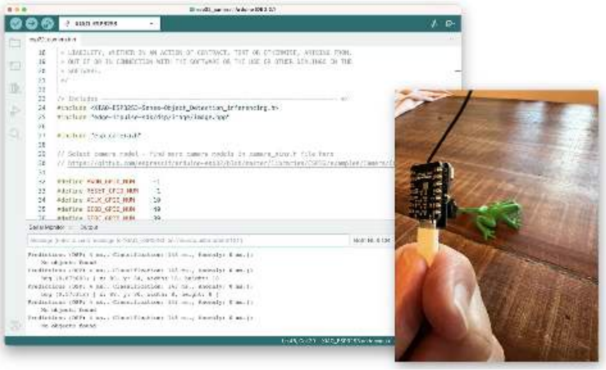
Fruits



```
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */

/* Includes ---------------------------------------------------------------- */
#include <XIAO-ESP32S3-Sense-Object_Detection_inferencing.h>
#include "edge-impulse-sdk/dsp/image/image.hpp"

#include "esp_camera.h"

// Select camera model - find more camera models in camera_pins.h file here
// https://github.com/espressif/arduino-esp32/blob/master/libraries/ESP32/examples/Camera/C

#define PWDN_GPIO_NUM     -1
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM     10
#define SIOD_GPIO_NUM     40
#define SIOC_GPIO_NUM     39
```

```
    fruit (0.[illegible]) [ x: 16, y: 32, width: 8, height: 8 ]
Predictions (DSP: 3 ms., Classification: 162 ms., Anomaly: 0 ms.):
    No objects found
Predictions (DSP: 3 ms., Classification: 162 ms., Anomaly: 0 ms.):
    fruit (0.[illegible]) [ x: 48, y: 32, width: 8, height: 8 ]
    fruit (0.[illegible]) [ x: 16, y: 32, width: 8, height: 8 ]
Predictions (DSP: 3 ms., Classification: 162 ms., Anomaly: 0 ms.):
    fruit (0.[illegible]) [ x: 24, y: 16, width: 8, height: 8 ]
Predictions (DSP: 3 ms., Classification: 162 ms., Anomaly: 0 ms.):
    fruit (0.[illegible]) [ x: 24, y: 16, width: 8, height: 8 ]
```

Bugs



```
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */

/* Includes ---------------------------------------------------------------- */
#include <XIAO-ESP32S3-Sense-Object_Detection_inferencing.h>
#include "edge-impulse-sdk/dsp/image/image.hpp"

#include "esp_camera.h"

// Select camera model - find more camera models in camera_pins.h file here
// https://github.com/espressif/arduino-esp32/blob/master/libraries/ESP32/examples/Camera/C

#define PWDN_GPIO_NUM     -1
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM     10
#define SIOD_GPIO_NUM     40
#define SIOC_GPIO_NUM     39
```

```
Predictions (DSP: 4 ms., Classification: 163 ms., Anomaly: 0 ms.):
    No objects found
Predictions (DSP: 4 ms., Classification: 163 ms., Anomaly: 0 ms.):
    bug (0.[illegible]) [ x: 30, y: 24, width: 16, height: 16 ]
Predictions (DSP: 4 ms., Classification: 163 ms., Anomaly: 0 ms.):
    bug (0.[illegible]) [ x: 40, y: 16, width: 8, height: 8 ]
Predictions (DSP: 4 ms., Classification: 163 ms., Anomaly: 0 ms.):
    No objects found
Predictions (DSP: 4 ms., Classification: 163 ms., Anomaly: 0 ms.):
    No objects found
```

Note that the model latency is 143 ms, and the frame rate per second is around 7 fps (similar to what we got with the Image Classification project). This happens because FOMO is cleverly built over a CNN model, not with an object detection model like the SSD MobileNet. For example, when running a MobileNetV2 SSD FPN-Lite 320 × 320 model on a Raspberry Pi 4, the latency is around five times higher (around 1.5 fps).

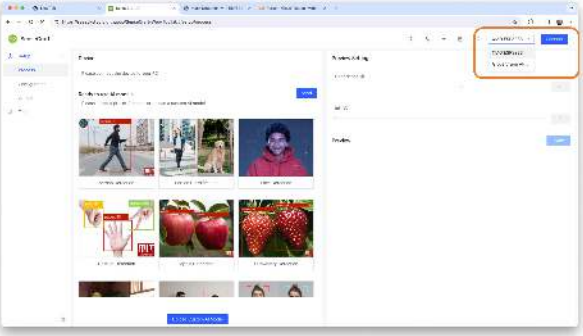
Deploying the Model (SenseCraft-Web-Toolkit)

As discussed in the Image Classification chapter, verifying inference with Image models on Arduino IDE is very challenging because we can not see what the camera focuses on. Again, let's use the **SenseCraft-Web Toolkit**.

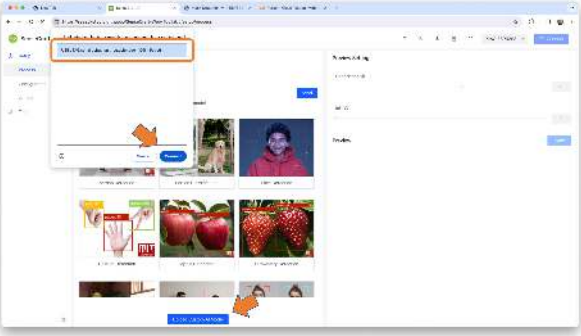
Follow the following steps to start the SenseCraft-Web-Toolkit:

1. Open the SenseCraft-Web-Toolkit website.
2. Connect the XIAO to your computer:

- Having the XIAO connected, select it as below:



- Select the device/Port and press `[Connect]`:



You can try several Computer Vision models previously uploaded by Seeed Studio. Try them and have fun!

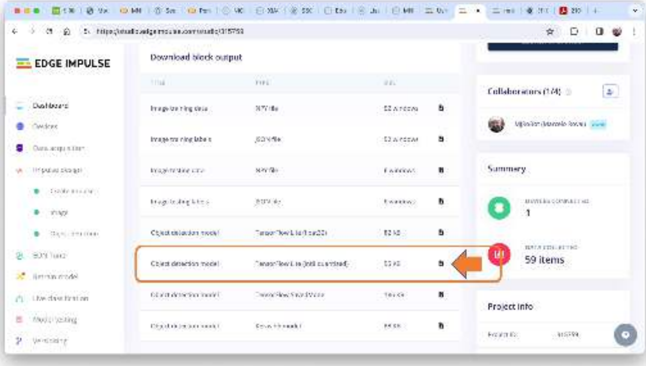
In our case, we will use the blue button at the bottom of the page: `[Upload Custom AI Model]`.

But first, we must download from Edge Impulse Studio our **quantized .tflite** model.

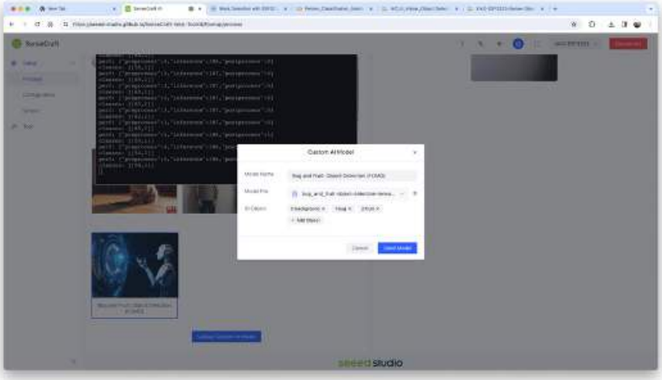
3. Go to your project at Edge Impulse Studio, or clone this one:

- XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v1-ESP-NN

4. On `Dashboard`, download the model ("block output"): `Object Detection model - TensorFlow Lite (int8 quantized)`

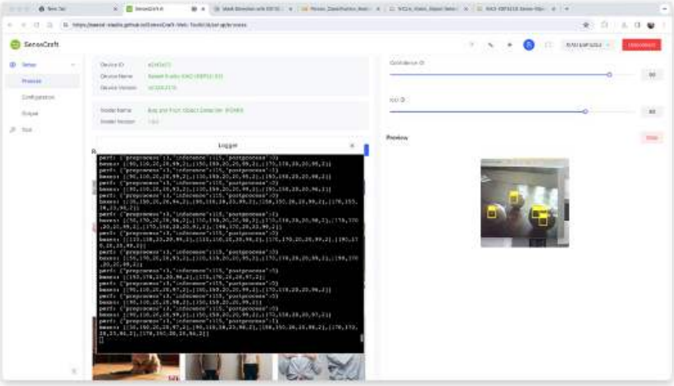


5. On SenseCraft-Web-Toolkit, use the blue button at the bottom of the page: `[Upload Custom AI Model]`. A window will pop up. Enter the Model file that you downloaded to your computer from Edge Impulse Studio, choose a Model Name, and enter with labels (ID: Object):



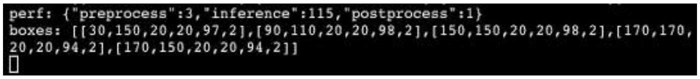
Note that you should use the labels trained on EI Studio and enter them in alphabetic order (in our case, background, bug, fruit).

After a few seconds (or minutes), the model will be uploaded to your device, and the camera image will appear in real-time on the Preview Sector:



The detected objects will be marked (the centroid). You can select the Confidence of your inference cursor `Confidence` and `IoU`, which is used to assess the accuracy of predicted bounding boxes compared to truth bounding boxes.

Clicking on the top button (Device Log), you can open a Serial Monitor to follow the inference, as we did with the Arduino IDE.



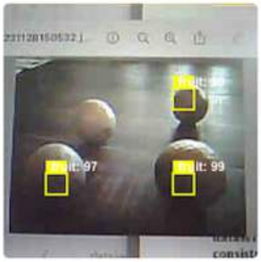
```
perf: {"preprocess":3,"inference":115,"postprocess":1}
boxes: [[30,150,20,20,97,2],[90,110,20,20,98,2],[150,150,20,20,98,2],[170,170,
20,20,94,2],[170,150,20,20,94,2]]
```

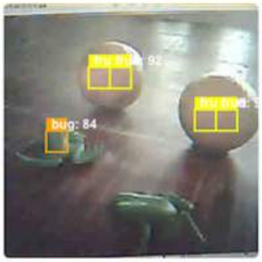
On Device Log, you will get information as:

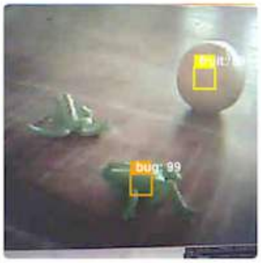
- Preprocess time (image capture and Crop): 3 ms,
- Inference time (model latency): 115 ms,
- Postprocess time (display of the image and marking objects): 1 ms.
- Output tensor (boxes), for example, one of the boxes: [[30,150, 20, 20,97, 2]]; where 30,150, 20, 20 are the coordinates of the box (around the centroid); 97 is the inference result, and 2 is the class (in this case 2: fruit).

  Note that in the above example, we got 5 boxes because none of the fruits got 3 centroids. One solution will be post-processing, where we can aggregate close centroids in one.

Here are other screenshots:







Conclusion

FOMO is a significant leap in the image processing space, as Louis Moreau and Mat Kelcey put it during its launch in 2022:

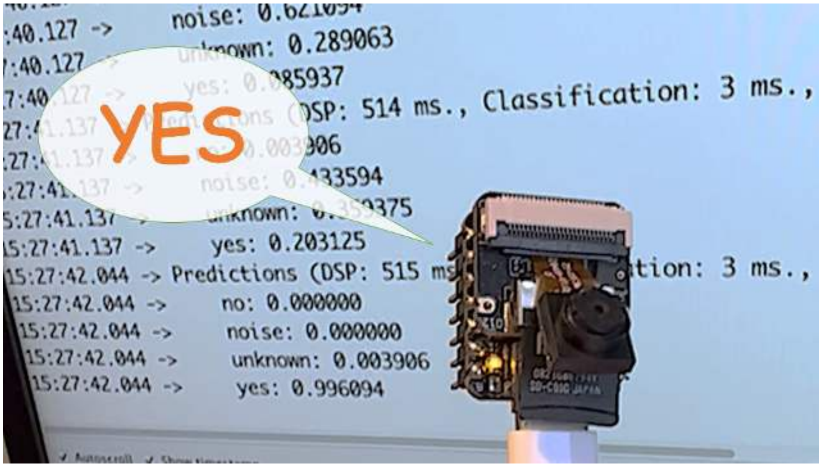
> FOMO is a ground-breaking algorithm that brings real-time object detection, tracking, and counting to microcontrollers for the first time.

Multiple possibilities exist for exploring object detection (and, more precisely, counting them) on embedded devices.

Resources

- Edge Impulse Project

Keyword Spotting (KWS)



```
:40.127 ->      noise: 0.621094
:40.127 ->      unknown: 0.289063
7:40.127 ->      yes: 0.085937
27:41.137 -> Predictions (DSP: 514 ms., Classification: 3 ms.,
:27:41.137 ->      no: 0.003906
:27:41.137 ->      noise: 0.433594
5:27:41.137 ->      unknown: 0.359375
15:27:41.137 ->      yes: 0.203125
15:27:42.044 -> Predictions (DSP: 515 ms                 tion: 3 ms.,
15:27:42.044 ->      no: 0.000000
15:27:42.044 ->      noise: 0.000000
15:27:42.044 ->      unknown: 0.003906
15:27:42.044 ->      yes: 0.996094
```

Figure 20.12: *Image by Marcelo Rovai*

Overview

Keyword Spotting (KWS) is integral to many voice recognition systems, enabling devices to respond to specific words or phrases. While this technology underpins popular devices like Google Assistant or Amazon Alexa, it's equally applicable and achievable on smaller, low-power devices. This lab will guide you through implementing a KWS system using TinyML on the XIAO ESP32S3 microcontroller board.

The XIAO ESP32S3, equipped with Espressif's ESP32-S3 chip, is a compact and potent microcontroller offering a dual-core Xtensa LX7 processor, integrated Wi-Fi, and Bluetooth. Its balance of computational power, energy efficiency, and versatile connectivity make it a fantastic platform for TinyML applications. Also, with its expansion board, we will have access to the "sense" part of the device, which has a 1600 × 1200 OV2640 camera, an SD card slot, and a **digital microphone**. The integrated microphone and the SD card will be essential in this project.

We will use the Edge Impulse Studio, a powerful, user-friendly platform that simplifies creating and deploying machine learning models onto edge devices.

We'll train a KWS model step-by-step, optimizing and deploying it onto the XIAO ESP32S3 Sense.

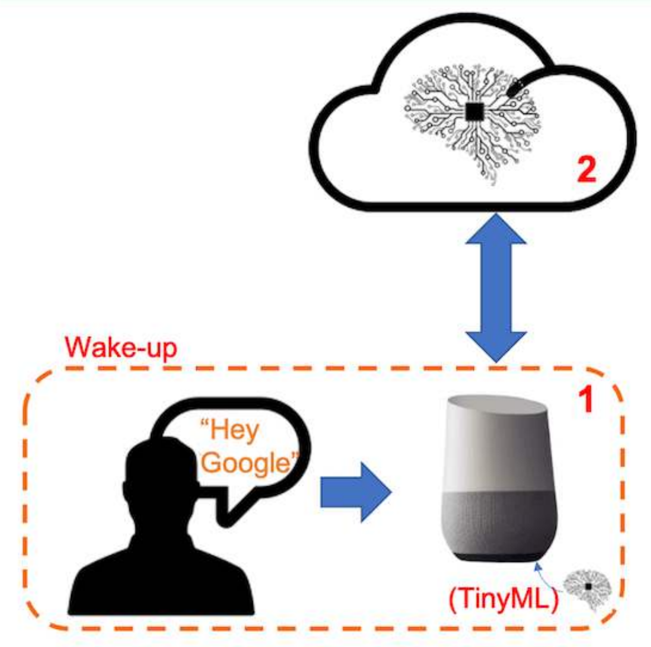
Our model will be designed to recognize keywords that can trigger device wake-up or specific actions (in the case of "YES"), bringing your projects to life with voice-activated commands.

Leveraging our experience with TensorFlow Lite for Microcontrollers (the engine "under the hood" on the EI Studio), we'll create a KWS system capable of real-time machine learning on the device.

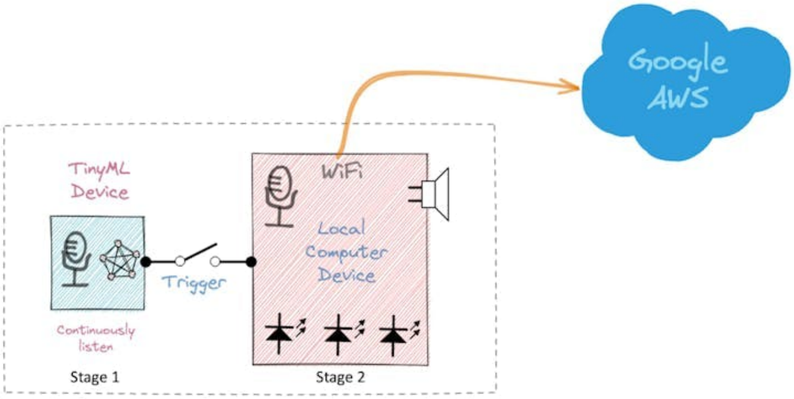
As we progress through the lab, we'll break down each process stage – from data collection and preparation to model training and deployment – to provide a comprehensive understanding of implementing a KWS system on a microcontroller.

How does a voice assistant work?

Keyword Spotting (KWS) is critical to many voice assistants, enabling devices to respond to specific words or phrases. To start, it is essential to realize that Voice Assistants on the market, like Google Home or Amazon Echo-Dot, only react to humans when they are "waked up" by particular keywords such as " Hey Google" on the first one and "Alexa" on the second.



In other words, recognizing voice commands is based on a multi-stage model or Cascade Detection.



Stage 1: A smaller microprocessor inside the Echo Dot or Google Home **continuously** listens to the sound, waiting for the keyword to be spotted. For such detection, a TinyML model at the edge is used (KWS application).

Stage 2: Only when triggered by the KWS application on Stage 1 is the data sent to the cloud and processed on a larger model.

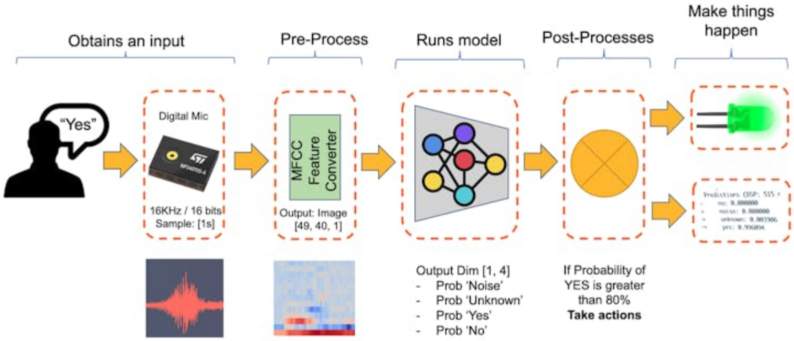
The video below shows an example where I emulate a Google Assistant on a Raspberry Pi (Stage 2), having an Arduino Nano 33 BLE as the tinyML device (Stage 1).

> If you want to go deeper on the full project, please see my tutorial: Building an Intelligent Voice Assistant From Scratch.

In this lab, we will focus on Stage 1 (KWS or Keyword Spotting), where we will use the XIAO ESP2S3 Sense, which has a digital microphone for spotting the keyword.

The KWS Project

The below diagram will give an idea of how the final KWS application should work (during inference):



```
Predictions (DSP: 515 ms
    no: 0.000000
    noise: 0.000000
    unknown: 0.003906
    yes: 0.996094
```

Our KWS application will recognize four classes of sound:

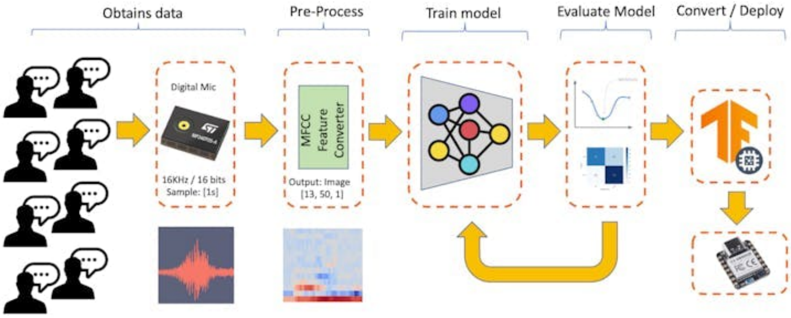
- **YES** (Keyword 1)

- **NO** (Keyword 2)
- **NOISE** (no keywords spoken, only background noise is present)
- **UNKNOWN** (a mix of different words than YES and NO)

> Optionally for real-world projects, it is always advised to include different words than keywords, such as "Noise" (or Background) and "Unknown."

The Machine Learning workflow

The main component of the KWS application is its model. So, we must train such a model with our specific keywords, noise, and other words (the "unknown"):



Dataset

The critical component of Machine Learning Workflow is the **dataset**. Once we have decided on specific keywords (*YES* and NO), we can take advantage of the dataset developed by Pete Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition." This dataset has 35 keywords (with +1,000 samples each), such as yes, no, stop, and go. In other words, we can get 1,500 samples of *yes* and *no*.

You can download a small portion of the dataset from Edge Studio (Keyword spotting pre-built dataset), which includes samples from the four classes we will use in this project: yes, no, noise, and background. For this, follow the steps below:

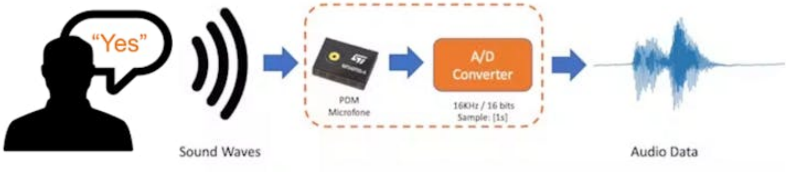
- Download the keywords dataset.
- Unzip the file in a location of your choice.

Although we have a lot of data from Pete's dataset, collecting some words spoken by us is advised. When working with accelerometers, creating a dataset with data captured by the same type of sensor was essential. In the case of *sound*, it is different because what we will classify is, in reality, *audio* data.

> The key difference between sound and audio is their form of energy. Sound is mechanical wave energy (longitudinal sound waves) that propagate through a medium causing variations in pressure within the medium. Audio is made of electrical energy (analog or digital signals) that represent sound electrically.

The sound waves should be converted to audio data when we speak a keyword. The conversion should be done by sampling the signal generated by the microphone in 16 kHz with a 16-bit depth.

So, any device that can generate audio data with this basic specification (16 kHz/16 bits) will work fine. As a device, we can use the proper XIAO ESP32S3 Sense, a computer, or even your mobile phone.



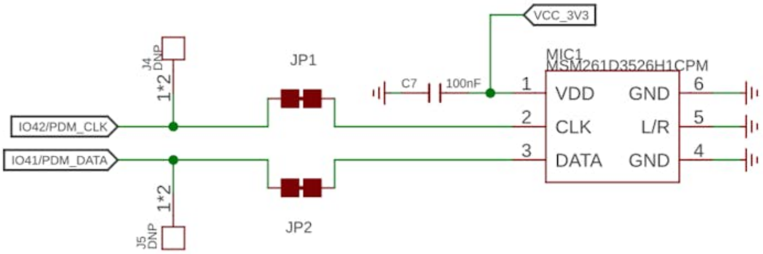
Capturing online Audio Data with Edge Impulse and a smartphone

In the lab Motion Classification and Anomaly Detection, we connect our device directly to Edge Impulse Studio for data capturing (having a sampling frequency of 50 Hz to 100 Hz). For such low frequency, we could use the EI CLI function *Data Forwarder,* but according to Jan Jongboom, Edge Impulse CTO, *audio* (16 kHz) *goes too fast for the data forwarder to be captured.* So, once we have the digital data captured by the microphone, we can turn *it into a WAV file* to be sent to the Studio via Data Uploader (same as we will do with Pete's dataset).

> If we want to collect audio data directly on the Studio, we can use any smartphone connected online with it. We will not explore this option here, but you can easily follow EI documentation.

Capturing (offline) Audio Data with the XIAO ESP32S3 Sense

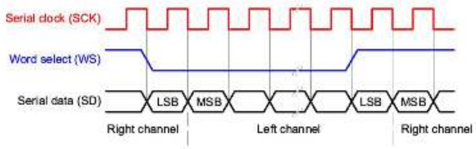
The built-in microphone is the MSM261D3526H1CPM, a PDM digital output MEMS microphone with Multi-modes. Internally, it is connected to the ESP32S3 via an I2S bus using pins IO41 (Clock) and IO41 (Data).



What is I2S?

I2S, or Inter-IC Sound, is a standard protocol for transmitting digital audio from one device to another. It was initially developed by Philips Semiconductor (now NXP Semiconductors). It is commonly used in audio devices such as digital signal processors, digital audio processors, and, more recently, microcontrollers with digital audio capabilities (our case here).

The I2S protocol consists of at least three lines:



1. Bit (or Serial) clock line (BCLK or CLK): This line toggles to indicate the start of a new bit of data (pin IO42).

2. Word select line (WS): This line toggles to indicate the start of a new word (left channel or right channel). The Word select clock (WS) frequency defines the sample rate. In our case, L/R on the microphone is set to ground, meaning that we will use only the left channel (mono).

3. Data line (SD): This line carries the audio data (pin IO41)

In an I2S data stream, the data is sent as a sequence of frames, each containing a left-channel word and a right-channel word. This makes I2S particularly suited for transmitting stereo audio data. However, it can also be used for mono or multichannel audio with additional data lines.

Let's start understanding how to capture raw data using the microphone. Go to the GitHub projectand download the sketch: XIAOEsp2s3_Mic_Test:

```
/*
  XIAO ESP32S3 Simple Mic Test
*/

#include <I2S.h>

void setup() {
  Serial.begin(115200);
  while (!Serial) {
  }

  // start I2S at 16 kHz with 16-bits per sample
  I2S.setAllPins(-1, 42, 41, -1, -1);
  if (!I2S.begin(PDM_MONO_MODE, 16000, 16)) {
    Serial.println("Failed to initialize I2S!");
    while (1); // do nothing
  }
}

void loop() {
  // read a sample
  int sample = I2S.read();

  if (sample && sample != -1 && sample != 1) {
    Serial.println(sample);
  }
}
```

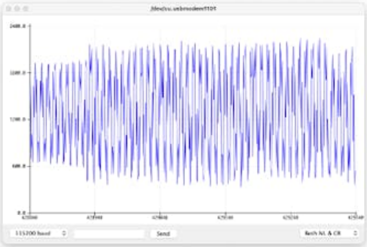
This code is a simple microphone test for the XIAO ESP32S3 using the I2S (Inter-IC Sound) interface. It sets up the I2S interface to capture audio data at a sample rate of 16 kHz with 16 bits per sample and then continuously reads samples from the microphone and prints them to the serial monitor.

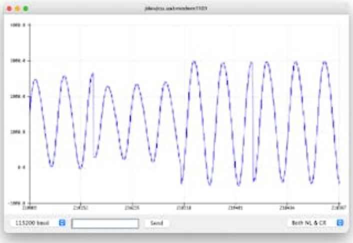
Let's dig into the code's main parts:

- Include the I2S library: This library provides functions to configure and use the I2S interface, which is a standard for connecting digital audio devices.
- I2S.setAllPins(–1, 42, 41, –1, –1): This sets up the I2S pins. The parameters are (–1, 42, 41, –1, –1), where the second parameter (42) is the PIN for the I2S clock (CLK), and the third parameter (41) is the PIN for the I2S data (DATA) line. The other parameters are set to –1, meaning those pins are not used.
- I2S.begin(PDM_MONO_MODE, 16000, 16): This initializes the I2S interface in Pulse Density Modulation (PDM) mono mode, with a sample rate of 16 kHz and 16 bits per sample. If the initialization fails, an error message is printed, and the program halts.
- int sample = I2S.read(): This reads an audio sample from the I2S interface.

If the sample is valid, it is printed on the Serial Monitor and Plotter.

Below is a test "whispering" in two different tones.



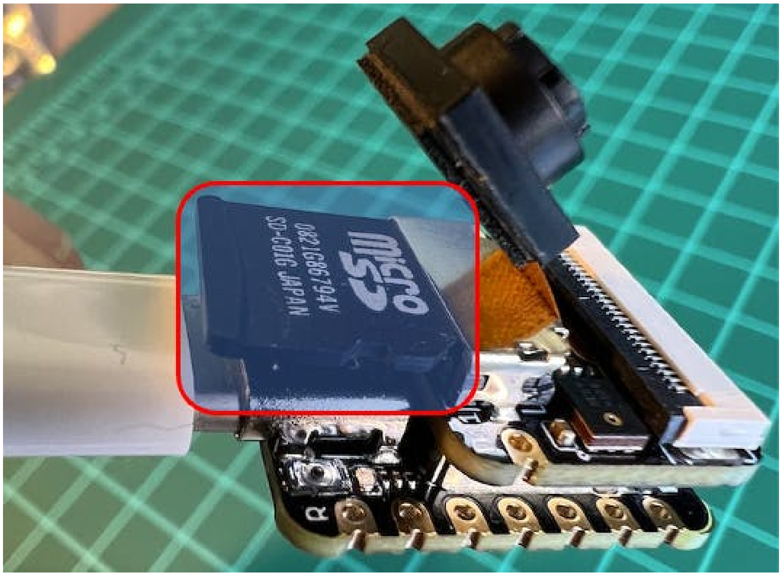


Save recorded sound samples (dataset) as .wav audio files to a microSD card

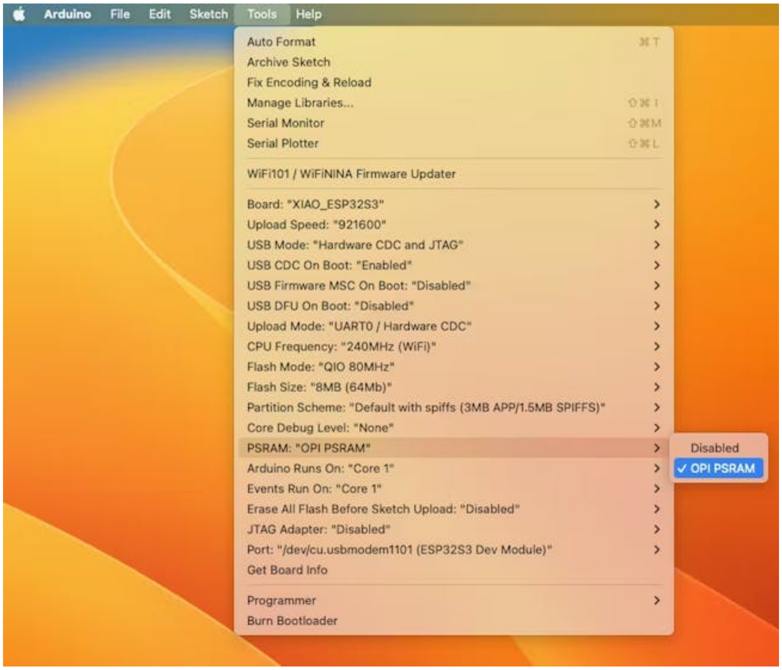
Let's use the onboard SD Card reader to save .wav audio files; we must habilitate the XIAO PSRAM first.

> ESP32-S3 has only a few hundred kilobytes of internal RAM on the MCU chip. It can be insufficient for some purposes so that ESP32-S3 can use up to 16 MB of external PSRAM (Psuedostatic RAM) connected in parallel with the SPI flash chip. The external memory is incorporated in the memory map and, with certain restrictions, is usable in the same way as internal data RAM.

For a start, Insert the SD Card on the XIAO as shown in the photo below (the SD Card should be formatted to FAT32).



Turn the PSRAM function of the ESP-32 chip on (Arduino IDE): Tools>PSRAM: “OPI PSRAM”>OPI PSRAM



Auto Format ⌘T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... ⇧⌘I
Serial Monitor ⇧⌘M
Serial Plotter ⇧⌘L

WiFi101 / WiFiNINA Firmware Updater

Board: "XIAO_ESP32S3"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
USB CDC On Boot: "Enabled"
USB Firmware MSC On Boot: "Disabled"
USB DFU On Boot: "Disabled"
Upload Mode: "UART0 / Hardware CDC"
CPU Frequency: "240MHz (WiFi)"
Flash Mode: "QIO 80MHz"
Flash Size: "8MB (64Mb)"
Partition Scheme: "Default with spiffs (3MB APP/1.5MB SPIFFS)"
Core Debug Level: "None"
PSRAM: "OPI PSRAM" → Disabled / ✓ OPI PSRAM
Arduino Runs On: "Core 1"
Events Run On: "Core 1"
Erase All Flash Before Sketch Upload: "Disabled"
JTAG Adapter: "Disabled"
Port: "/dev/cu.usbmodem1101 (ESP32S3 Dev Module)"
Get Board Info

Programmer
Burn Bootloader

- Download the sketch Wav_Record_dataset, which you can find on the project's GitHub.

This code records audio using the I2S interface of the Seeed XIAO ESP32S3 Sense board, saves the recording as a.wav file on an SD card, and allows for control of the recording process through commands sent from the serial monitor. The name of the audio file is customizable (it should be the class labels to be used with the training), and multiple recordings can be made, each saved in a new file. The code also includes functionality to increase the volume of the recordings.

Let's break down the most essential parts of it:

```
#include <I2S.h>
#include "FS.h"
#include "SD.h"
#include "SPI.h"
```

Those are the necessary libraries for the program. I2S.h allows for audio input, FS.h provides file system handling capabilities, SD.h enables the program to interact with an SD card, and SPI.h handles the SPI communication with the SD card.

```
#define RECORD_TIME   10
#define SAMPLE_RATE 16000U
#define SAMPLE_BITS 16
#define WAV_HEADER_SIZE 44
#define VOLUME_GAIN 2
```

Here, various constants are defined for the program.

- **RECORD_TIME** specifies the length of the audio recording in seconds.
- **SAMPLE_RATE** and **SAMPLE_BITS** define the audio quality of the recording.
- **WAV_HEADER_SIZE** specifies the size of the .wav file header.
- **VOLUME_GAIN** is used to increase the volume of the recording.

```
int fileNumber = 1;
String baseFileName;
bool isRecording = false;
```

These variables keep track of the current file number (to create unique file names), the base file name, and whether the system is currently recording.

```
void setup() {
  Serial.begin(115200);
  while (!Serial);

  I2S.setAllPins(-1, 42, 41, -1, -1);
  if (!I2S.begin(PDM_MONO_MODE, SAMPLE_RATE, SAMPLE_BITS)) {
    Serial.println("Failed to initialize I2S!");
    while (1);
  }

  if(!SD.begin(21)){
    Serial.println("Failed to mount SD Card!");
    while (1);
  }
  Serial.printf("Enter with the label name\n");
}
```

The setup function initializes the serial communication, I2S interface for audio input, and SD card interface. If the I2S did not initialize or the SD card fails to mount, it will print an error message and halt execution.

```
void loop() {
  if (Serial.available() > 0) {
    String command = Serial.readStringUntil('\n');
    command.trim();
    if (command == "rec") {
      isRecording = true;
    } else {
      baseFileName = command;
      fileNumber = 1; //reset file number each time a new
                        basefile name is set
      Serial.printf("Send rec for starting recording label \n");
    }
  }
  if (isRecording && baseFileName != "") {
    String fileName = "/" + baseFileName + "."
                      + String(fileNumber) + ".wav";
    fileNumber++;
    record_wav(fileName);
    delay(1000); // delay to avoid recording multiple files
                  at once
    isRecording = false;
  }
}
```

In the main loop, the program waits for a command from the serial monitor. If the command is rec, the program starts recording. Otherwise, the command is assumed to be the base name for the .wav files. If it's currently recording and a base file name is set, it records the audio and saves it as a.wav file. The file names are generated by appending the file number to the base file name.

```
void record_wav(String fileName)
{
  ...

  File file = SD.open(fileName.c_str(), FILE_WRITE);
  ...
  rec_buffer = (uint8_t *)ps_malloc(record_size);
  ...

  esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,
                    rec_buffer,
                    record_size,
                    &sample_size,
                    portMAX_DELAY);
  ...
}
```

This function records audio and saves it as a.wav file with the given name. It starts by initializing the sample_size and record_size variables. record_size is calculated based on the sample rate, size, and desired recording time. Let's dig into the essential sections;

```
File file = SD.open(fileName.c_str(), FILE_WRITE);
// Write the header to the WAV file
uint8_t wav_header[WAV_HEADER_SIZE];
generate_wav_header(wav_header, record_size, SAMPLE_RATE);
file.write(wav_header, WAV_HEADER_SIZE);
```

This section of the code opens the file on the SD card for writing and then generates the .wav file header using the generate_wav_header function. It then writes the header to the file.

```
// PSRAM malloc for recording
rec_buffer = (uint8_t *)ps_malloc(record_size);
if (rec_buffer == NULL) {
  Serial.printf("malloc failed!\n");
  while(1) ;
}
Serial.printf("Buffer: %d bytes\n", ESP.getPsramSize()
               - ESP.getFreePsram());
```

The ps_malloc function allocates memory in the PSRAM for the recording. If the allocation fails (i.e., rec_buffer is NULL), it prints an error message and halts execution.

```
// Start recording
esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,
         rec_buffer,
         record_size,
         &sample_size,
         portMAX_DELAY);
if (sample_size == 0) {
  Serial.printf("Record Failed!\n");
} else {
    Serial.printf("Record %d bytes\n", sample_size);
  }
```

The i2s_read function reads audio data from the microphone into rec_buffer. It prints an error message if no data is read (sample_size is 0).

```
// Increase volume
for (uint32_t i = 0; i < sample_size; i += SAMPLE_BITS/8) {
  (*(uint16_t *)(rec_buffer+i)) <<= VOLUME_GAIN;
}
```

This section of the code increases the recording volume by shifting the sample values by VOLUME_GAIN.

```
// Write data to the WAV file
Serial.printf("Writing to the file ...\n");
if (file.write(rec_buffer, record_size) != record_size)
  Serial.printf("Write file Failed!\n");

free(rec_buffer);
file.close();
Serial.printf("Recording complete: \n");
Serial.printf("Send rec for a new sample or enter
                a new label\n\n");
```

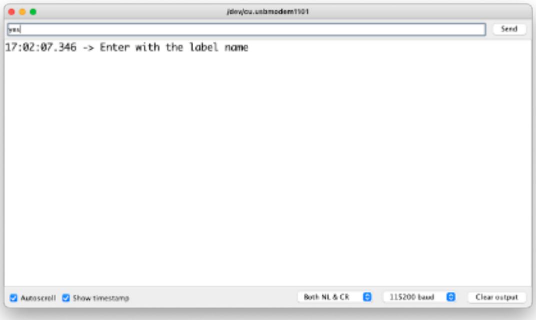
Finally, the audio data is written to the .wav file. If the write operation fails, it prints an error message. After writing, the memory allocated for rec_buffer is freed, and the file is closed. The function finishes by printing a completion message and prompting the user to send a new command.

```
void generate_wav_header(uint8_t *wav_header,
             uint32_t wav_size,
             uint32_t sample_rate)
{
  ...
  memcpy(wav_header, set_wav_header, sizeof(set_wav_header));
}
```

The generate_wav_header function creates a.wav file header based on the parameters (wav_size and sample_rate). It generates an array of bytes according to the .wav file format, which includes fields for the file size, audio format, number of channels, sample rate, byte rate, block alignment, bits per sample, and data size. The generated header is then copied into the wav_header array passed to the function.

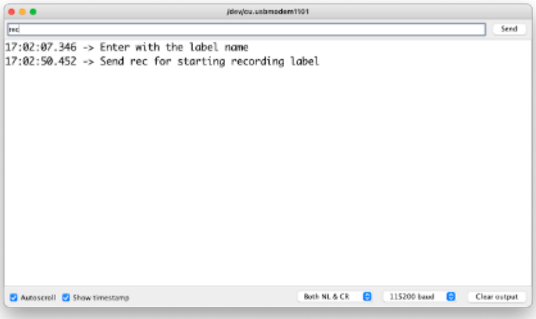
Now, upload the code to the XIAO and get samples from the keywords (yes and no). You can also capture noise and other words.

The Serial monitor will prompt you to receive the label to be recorded.



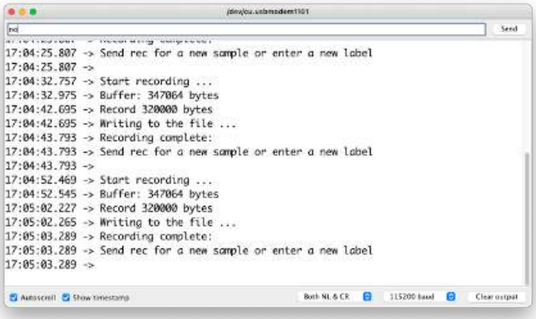
```
17:02:07.346 -> Enter with the label name
```

Send the label (for example, yes). The program will wait for another command: rec



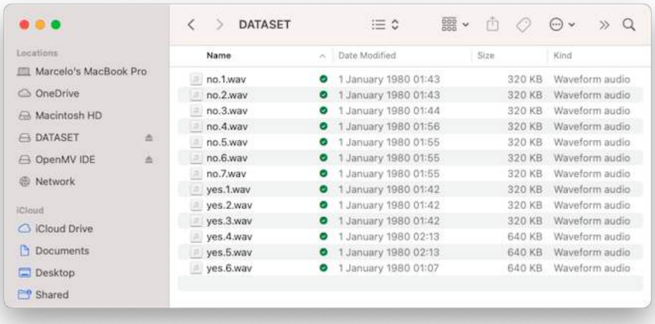
```
17:02:07.346 -> Enter with the label name
17:02:50.452 -> Send rec for starting recording label
```

And the program will start recording new samples every time a command rec is sent. The files will be saved as yes.1.wav, yes.2.wav, yes.3.wav, etc., until a new label (for example, no) is sent. In this case, you should send the command rec for each new sample, which will be saved as no.1.wav, no.2.wav, no.3.wav, etc.



```
17:04:25.807 -> Send rec for a new sample or enter a new label
17:04:25.807 ->
17:04:32.757 -> Start recording ...
17:04:32.975 -> Buffer: 347064 bytes
17:04:42.695 -> Record 320000 bytes
17:04:42.695 -> Writing to the file ...
17:04:43.793 -> Recording complete:
17:04:43.793 -> Send rec for a new sample or enter a new label
17:04:43.793 ->
17:04:52.469 -> Start recording ...
17:04:52.545 -> Buffer: 347064 bytes
17:05:02.227 -> Record 320000 bytes
17:05:02.265 -> Writing to the file ...
17:05:03.289 -> Recording complete:
17:05:03.289 -> Send rec for a new sample or enter a new label
17:05:03.289 ->
```

Ultimately, we will get the saved files on the SD card.

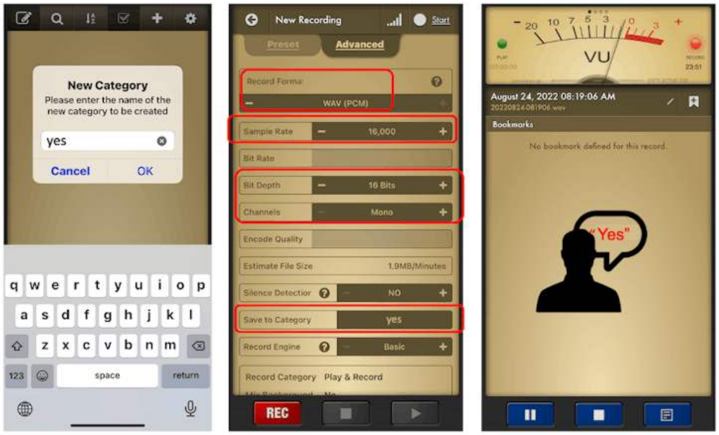


Name	Date Modified	Size	Kind
no.1.wav	1 January 1980 01:43	320 KB	Waveform audio
no.2.wav	1 January 1980 01:43	320 KB	Waveform audio
no.3.wav	1 January 1980 01:44	320 KB	Waveform audio
no.4.wav	1 January 1980 01:56	320 KB	Waveform audio
no.5.wav	1 January 1980 01:55	320 KB	Waveform audio
no.6.wav	1 January 1980 01:55	320 KB	Waveform audio
no.7.wav	1 January 1980 01:55	320 KB	Waveform audio
yes.1.wav	1 January 1980 01:42	320 KB	Waveform audio
yes.2.wav	1 January 1980 01:42	320 KB	Waveform audio
yes.3.wav	1 January 1980 01:42	320 KB	Waveform audio
yes.4.wav	1 January 1980 02:13	640 KB	Waveform audio
yes.5.wav	1 January 1980 02:13	640 KB	Waveform audio
yes.6.wav	1 January 1980 01:07	640 KB	Waveform audio

The files are ready to be uploaded to Edge Impulse Studio

Capturing (offline) Audio Data Apps

Alternatively, you can also use your PC or smartphone to capture audio data with a sampling frequency 16 kHz and a bit depth of 16 Bits. A good app for that is *Voice Recorder Pro* (IOS). You should save your records as .wav files and send them to your computer.



New Category
Please enter the name of the new category to be created
yes
Cancel OK

New Recording
Preset Advanced
Record Format: WAV (PCM)
Sample Rate: 16,000
Bit Rate
Bit Depth: 16 Bits
Channels: Mono
Encode Quality
Estimate File Size: 1.9MB/Minutes
Silence Detection: NO
Save to Category: yes
Record Engine: Basic
Record Category: Play & Record

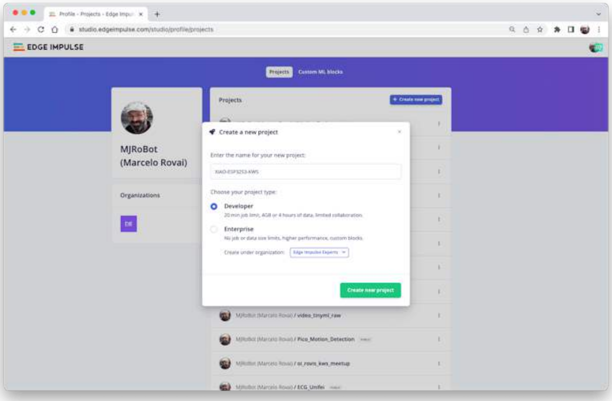
August 24, 2022 08:19:06 AM
20220824081906.wav
Bookmarks
No bookmark defined for this record.

Note that any app, such as Audacity, can be used for audio recording or even your computer.

Training model with Edge Impulse Studio

Uploading the Data

When the raw dataset is defined and collected (Pete's dataset + recorded keywords), we should initiate a new project at Edge Impulse Studio:



Create a new project

Enter the name for your new project:

XIAO-ESP32S3-KWS

Choose your project type:

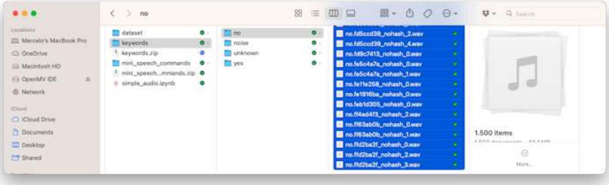
● Developer
20 min job limit, 4GB or 4 hours of data, limited collaboration.

○ Enterprise
No job or data size limits, higher performance, custom blocks.

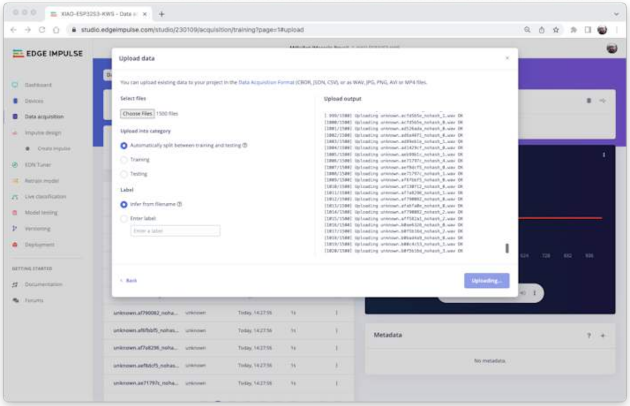
Create under organization: Edge Impulse Experts

Create new project

Once the project is created, select the Upload Existing Data tool in the Data Acquisition section. Choose the files to be uploaded:



And upload them to the Studio (You can automatically split data in train/test). Repeat to all classes and all raw data.



Upload data

You can upload existing data to your project in the Data Acquisition Format (CBOR, JSON, CSV), or as WAV, JPG, PNG, AVI or MP4 files.

Select files

Choose Files 1500 files

Upload into category

● Automatically split between training and testing
○ Training
○ Testing

Label

● Infer from filename
○ Enter label

Upload output

```
[ 999/1500] Uploading unknown.acfd565e_nohash_1.wav OK
[1000/1500] Uploading unknown.acfd565e_nohash_0.wav OK
[1001/1500] Uploading unknown.ad526ada_nohash_0.wav OK
[1002/1500] Uploading unknown.ad6a46f1_nohash_0.wav OK
[1003/1500] Uploading unknown.ad89eb1e_nohash_1.wav OK
[1004/1500] Uploading unknown.ad1429cf_nohash_0.wav OK
[1005/1500] Uploading unknown.aeb99b1c_nohash_1.wav OK
[1006/1500] Uploading unknown.ae71797c_nohash_4.wav OK
[1007/1500] Uploading unknown.aef9dcf5_nohash_0.wav OK
[1008/1500] Uploading unknown.ae71797c_nohash_1.wav OK
[1009/1500] Uploading unknown.af6fbbf5_nohash_0.wav OK
[1010/1500] Uploading unknown.af130f12_nohash_0.wav OK
[1011/1500] Uploading unknown.af7a8296_nohash_1.wav OK
[1012/1500] Uploading unknown.af790082_nohash_0.wav OK
[1013/1500] Uploading unknown.afabfa0e_nohash_0.wav OK
[1014/1500] Uploading unknown.af790082_nohash_2.wav OK
[1015/1500] Uploading unknown.aff582a1_nohash_2.wav OK
[1016/1500] Uploading unknown.b0ae6326_nohash_0.wav OK
[1017/1500] Uploading unknown.b0f5b16d_nohash_2.wav OK
[1018/1500] Uploading unknown.b0bad4a9_nohash_0.wav OK
[1019/1500] Uploading unknown.b00c4c53_nohash_1.wav OK
[1020/1500] Uploading unknown.b0f5616d_nohash_3.wav OK
```

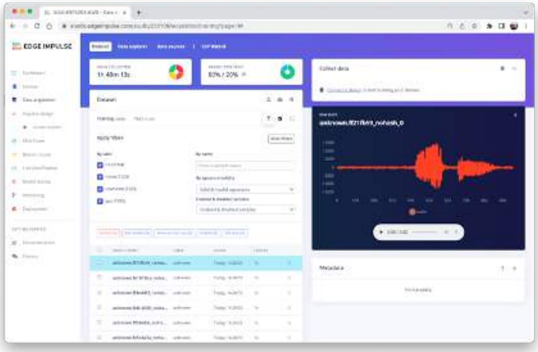
Uploading...

unknown.af790082_noha...	unknown	Today, 14:27:56	1s
unknown.af6fbbf5_nohas...	unknown	Today, 14:27:56	1s
unknown.af7a8296_noha...	unknown	Today, 14:27:56	1s
unknown.aef9dcf5_nohas...	unknown	Today, 14:27:56	1s
unknown.ae71797c_noha...	unknown	Today, 14:27:56	1s

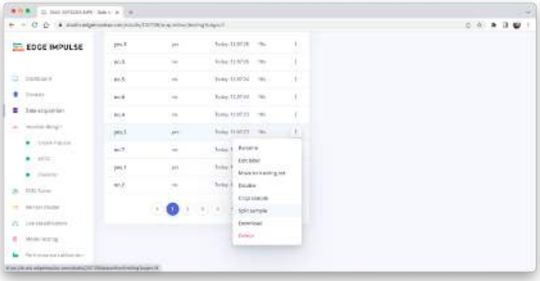
Metadata

No metadata.

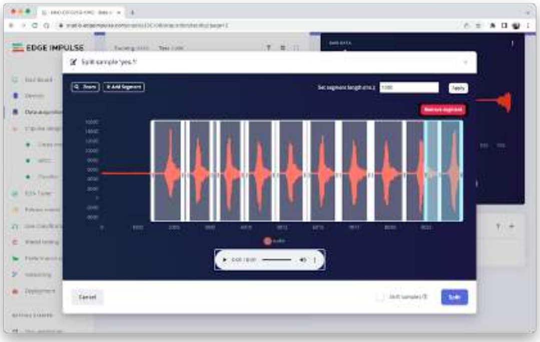
The samples will now appear in the Data acquisition section.



All data on Pete's dataset have a 1 s length, but the samples recorded in the previous section have 10 s and must be split into 1s samples to be compatible. Click on three dots after the sample name and select Split sample.



Once inside the tool, split the data into 1-second records. If necessary, add or remove segments:



This procedure should be repeated for all samples.

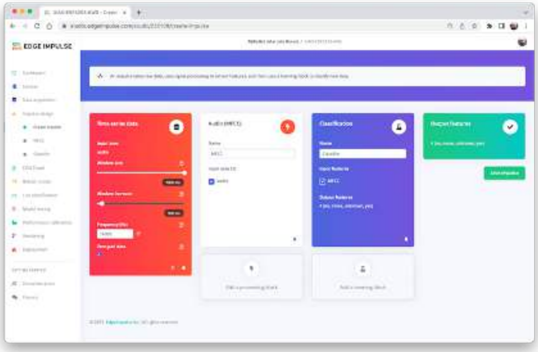
> Note: For longer audio files (minutes), first, split into 10-second segments and after that, use the tool again to get the final 1-second splits.

Suppose we do not split data automatically in train/test during upload. In that case, we can do it manually (using the three dots menu, moving samples individually) or using Perform Train / Test Split on Dashboard – Danger Zone.

We can optionally check all datasets using the tab Data Explorer.

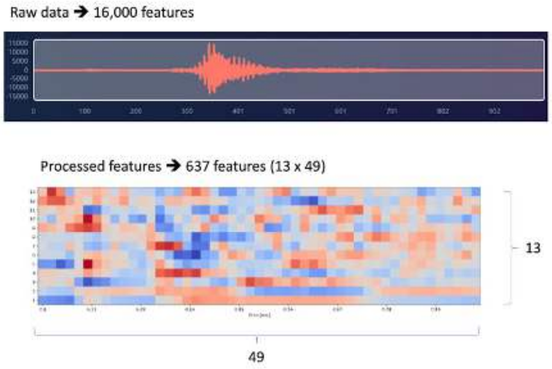
Creating Impulse (Pre-Process / Model definition)

An **impulse** *takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.*



First, we will take the data points with a 1-second window, augmenting the data, sliding that window each 500 ms. Note that the option zero-pad data is set. It is essential to fill with zeros samples smaller than 1 second (in some cases, I reduced the 1000 ms window on the split tool to avoid noises and spikes).

Each 1-second audio sample should be pre-processed and converted to an image (for example, 13 × 49 × 1). We will use MFCC, which extracts features from audio signals using Mel Frequency Cepstral Coefficients, which are great for the human voice.

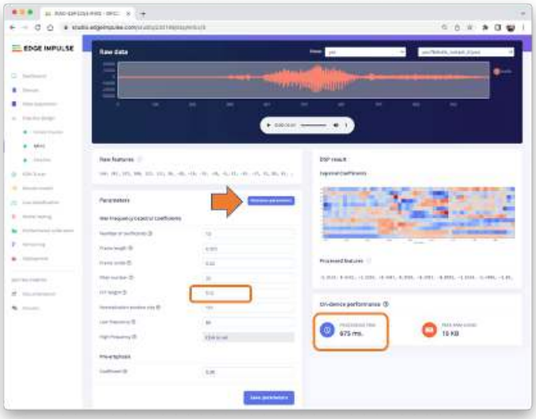


Next, we select KERAS for classification and build our model from scratch by doing Image Classification using Convolution Neural Network).

Pre-Processing (MFCC)

The next step is to create the images to be trained in the next phase:

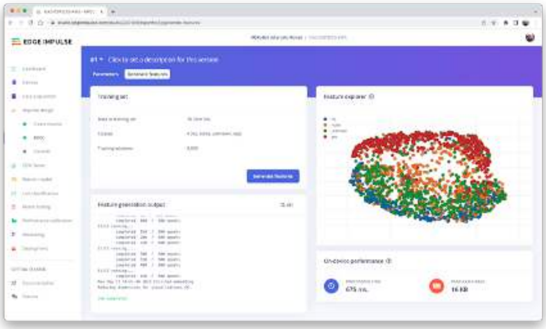
We can keep the default parameter values or take advantage of the DSP Autotuneparameters option, which we will do.



The result will not spend much memory to pre-process data (only 16KB). Still, the estimated processing time is high, 675 ms for an Espressif ESP-EYE (the closest reference available), with a 240 kHz clock (same as our device), but with a smaller CPU (XTensa LX6, versus the LX7 on the ESP32S). The real inference time should be smaller.

Suppose we need to reduce the inference time later. In that case, we should return to the pre-processing stage and, for example, reduce the FFT length to 256, change the Number of coefficients, or another parameter.

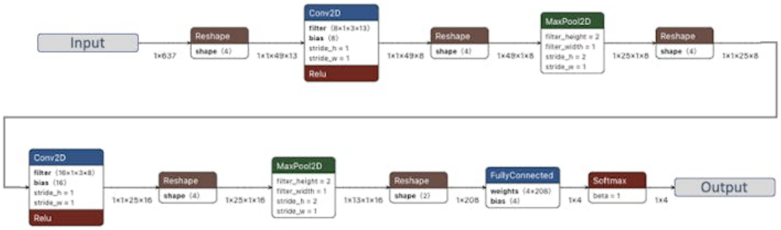
For now, let's keep the parameters defined by the Autotuning tool. Save parameters and generate the features.



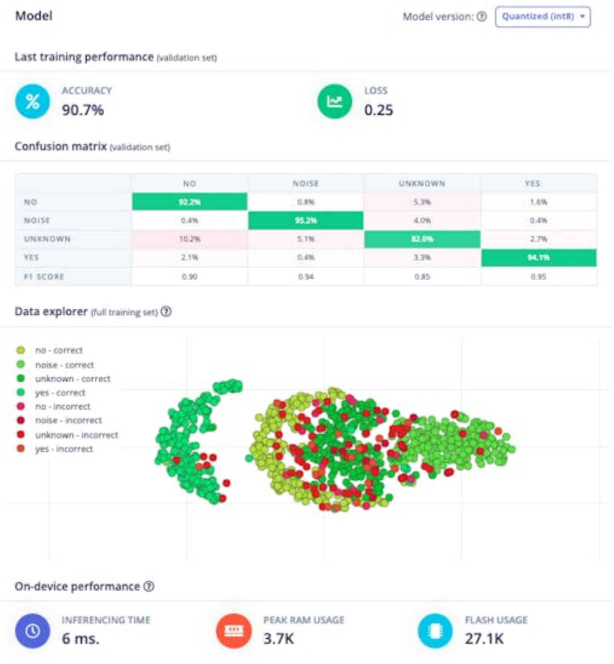
If you want to go further with converting temporal serial data into images using FFT, Spectrogram, etc., you can play with this CoLab: Audio Raw Data Analysis.

Model Design and Training

We will use a Convolution Neural Network (CNN) model. The basic architecture is defined with two blocks of Conv1D + MaxPooling (with 8 and 16 neurons, respectively) and a 0.25 Dropout. And on the last layer, after Flattening four neurons, one for each class:



As hyper-parameters, we will have a Learning Rate of 0.005 and a model that will be trained by 100 epochs. We will also include data augmentation, as some noise. The result seems OK:



Model

Model version: ⓘ Quantized (int8) ▾

Last training performance (validation set)

ACCURACY
90.7%

LOSS
0.25

Confusion matrix (validation set)

	NO	NOISE	UNKNOWN	YES
NO	92.2%	0.8%	5.3%	1.6%
NOISE	0.4%	95.2%	4.0%	0.4%
UNKNOWN	10.2%	5.1%	82.0%	2.7%
YES	2.1%	0.4%	3.3%	94.1%
F1 SCORE	0.90	0.94	0.85	0.95

Data explorer (full training set) ⓘ

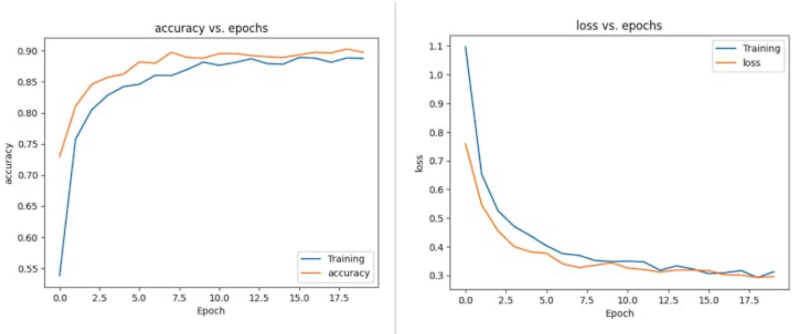
On-device performance ⓘ

INFERENCING TIME
6 ms.

PEAK RAM USAGE
3.7K

FLASH USAGE
27.1K

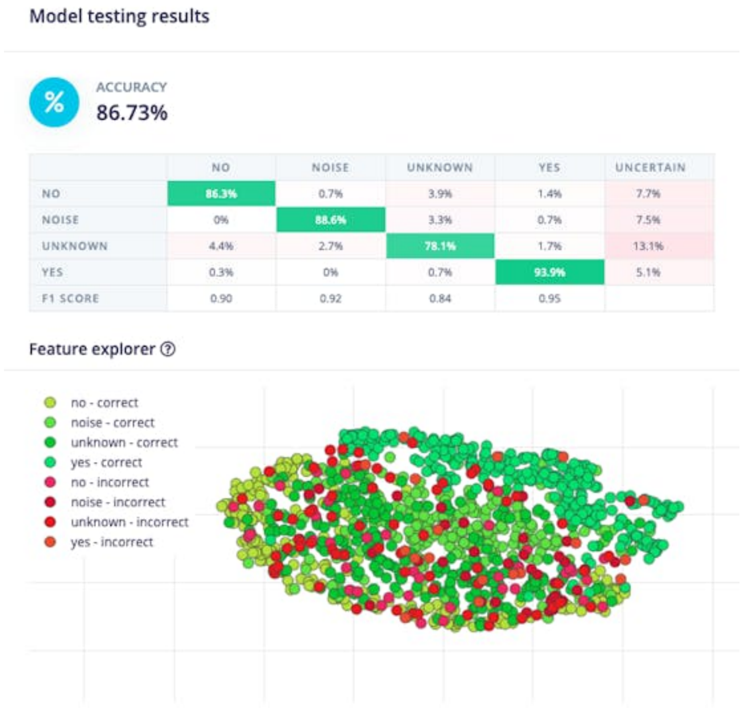
If you want to understand what is happening "under the hood," you can download the dataset and run a Jupyter Notebook playing with the code. For example, you can analyze the accuracy by each epoch:



This CoLab Notebook can explain how you can go further: KWS Classifier Project - Looking "Under the hood Training/xiao_esp32s3_keyword_spotting_-project_nn_classifier.ipynb)."

Testing

Testing the model with the data put apart before training (Test Data), we got an accuracy of approximately 87%.



Model testing results

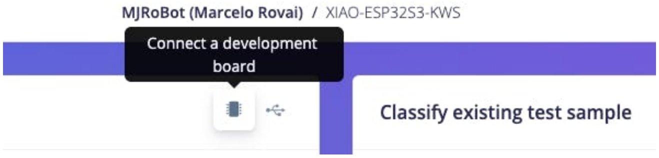
ACCURACY
86.73%

	NO	NOISE	UNKNOWN	YES	UNCERTAIN
NO	86.3%	0.7%	3.9%	1.4%	7.7%
NOISE	0%	88.6%	3.3%	0.7%	7.5%
UNKNOWN	4.4%	2.7%	78.1%	1.7%	13.1%
YES	0.3%	0%	0.7%	93.9%	5.1%
F1 SCORE	0.90	0.92	0.84	0.95	

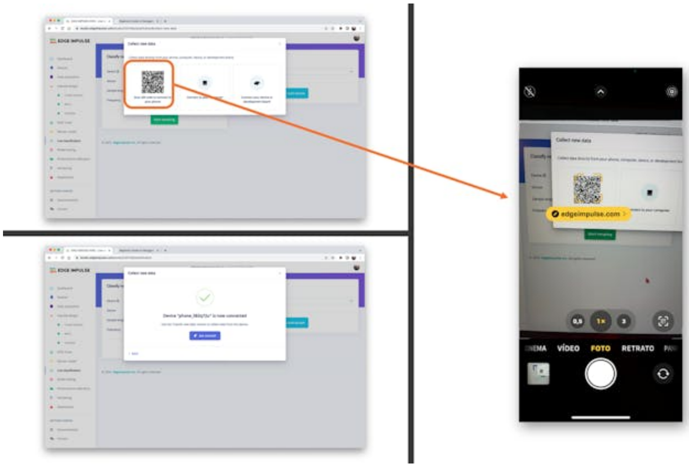
Feature explorer ⓘ

Inspecting the F1 score, we can see that for YES, we got 0.95, an excellent result once we used this keyword to "trigger" our postprocessing stage (turn on the built-in LED). Even for NO, we got 0.90. The worst result is for unknown, what is OK.

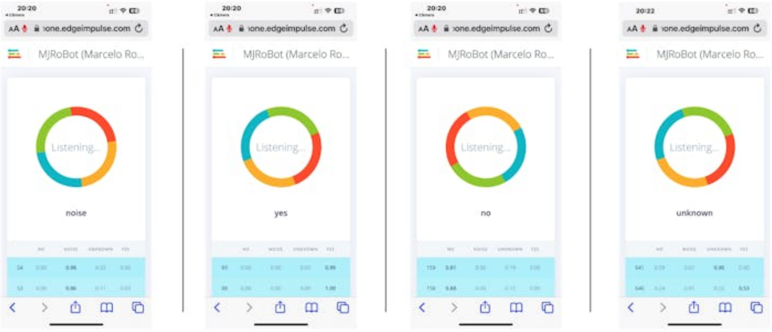
We can proceed with the project, but it is possible to perform Live Classification using a smartphone before deployment on our device. Go to the Live Classification section and click on Connect a Development board:


MJRoBot (Marcelo Rovai) / XIAO-ESP32S3-KWS
Connect a development board
Classify existing test sample

Point your phone to the barcode and select the link.

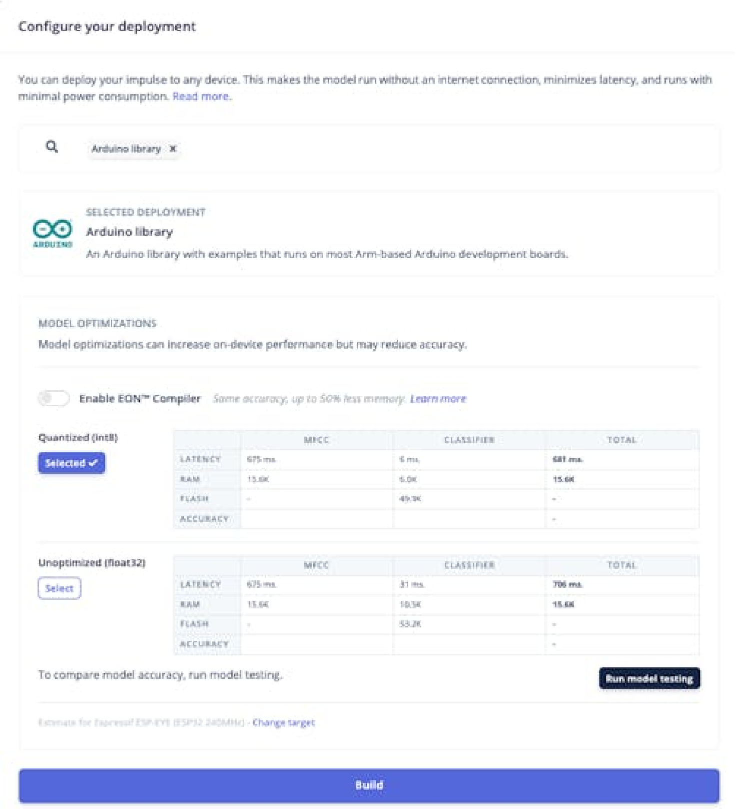


Your phone will be connected to the Studio. Select the option Classification on the app, and when it is running, start testing your keywords, confirming that the model is working with live and real data:



Deploy and Inference

The Studio will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. You should select the option Arduino Library, and at the bottom, choose Quantized (Int8) and press the button Build.



Configure your deployment

You can deploy your impulse to any device. This makes the model run without an internet connection, minimizes latency, and runs with minimal power consumption. Read more.

Arduino library ✕

SELECTED DEPLOYMENT
Arduino library
An Arduino library with examples that runs on most Arm-based Arduino development boards.

MODEL OPTIMIZATIONS
Model optimizations can increase on-device performance but may reduce accuracy.

○ Enable EON™ Compiler *Same accuracy, up to 50% less memory. Learn more*

Quantized (int8) Selected ✓

	MFCC	CLASSIFIER	TOTAL
LATENCY	675 ms.	6 ms.	681 ms.
RAM	15.6K	6.0K	15.6K
FLASH	-	49.9K	-
ACCURACY			-

Unoptimized (float32) Select

	MFCC	CLASSIFIER	TOTAL
LATENCY	675 ms.	31 ms.	706 ms.
RAM	15.6K	10.5K	15.6K
FLASH	-	53.2K	-
ACCURACY			-

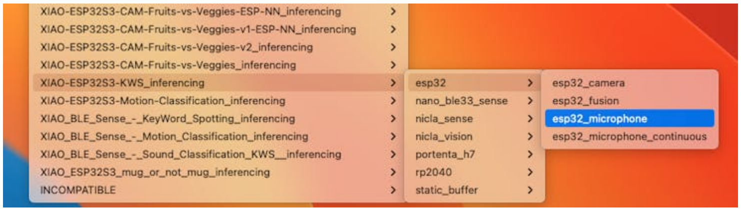
To compare model accuracy, run model testing. Run model testing

Estimate for Espressif ESP-EYE (ESP32 240MHz) - Change target

Build

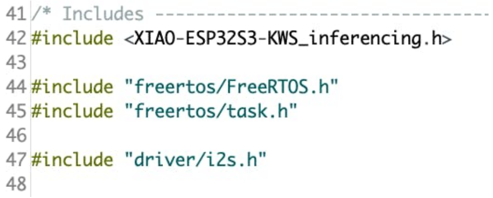
Now it is time for a real test. We will make inferences wholly disconnected from the Studio. Let's change one of the ESP32 code examples created when you deploy the Arduino Library.

In your Arduino IDE, go to the File/Examples tab look for your project, and select esp32/esp32_microphone:


XIAO-ESP32S3-CAM-Fruits-vs-Veggies-ESP-NN_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v1-ESP-NN_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v2_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing
XIAO-ESP32S3-KWS_inferencing
XIAO-ESP32S3-Motion-Classification_inferencing
XIAO_BLE_Sense_-_KeyWord_Spotting_inferencing
XIAO_BLE_Sense_-_Motion_Classification_inferencing
XIAO_BLE_Sense_-_Sound_Classification_KWS__inferencing
XIAO_ESP32S3_mug_or_not_mug_inferencing
INCOMPATIBLE
esp32
nano_ble33_sense
nicla_sense
nicla_vision
portenta_h7
rp2040
static_buffer
esp32_camera
esp32_fusion
esp32_microphone
esp32_microphone_continuous

This code was created for the ESP-EYE built-in microphone, which should be adapted for our device.

Start changing the libraries to handle the I2S bus:



```
/* Includes ---------------------------------
#include <XIAO-ESP32S3-KWS_inferencing.h>

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#include "driver/i2s.h"

```

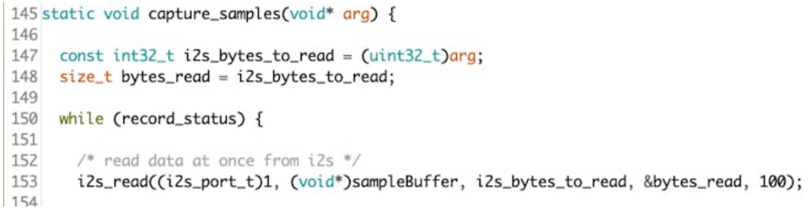
By:

```
#include <I2S.h>
#define SAMPLE_RATE 16000U
#define SAMPLE_BITS 16
```

Initialize the IS2 microphone at setup(), including the lines:

```
void setup()
{
...
    I2S.setAllPins(-1, 42, 41, -1, -1);
    if (!I2S.begin(PDM_MONO_MODE, SAMPLE_RATE, SAMPLE_BITS)) {
      Serial.println("Failed to initialize I2S!");
    while (1) ;
...
}
```

On the static void capture_samples(void* arg) function, replace the line 153 that reads data from I2S mic:

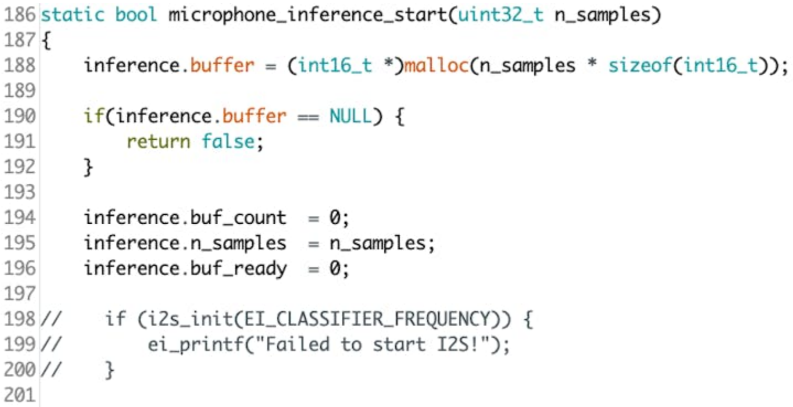


```
145 static void capture_samples(void* arg) {
146
147   const int32_t i2s_bytes_to_read = (uint32_t)arg;
148   size_t bytes_read = i2s_bytes_to_read;
149
150   while (record_status) {
151
152     /* read data at once from i2s */
153     i2s_read((i2s_port_t)1, (void*)sampleBuffer, i2s_bytes_to_read, &bytes_read, 100);
154
```

By:

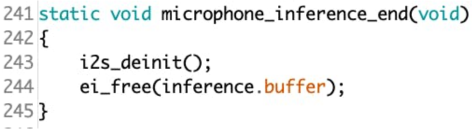
```
/* read data at once from i2s */
esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,
                 (void*)sampleBuffer,
                 i2s_bytes_to_read,
                 &bytes_read, 100);
```

On function static bool microphone_inference_start(uint32_t n_samples), we should comment or delete lines 198 to 200, where the microphone initialization function is called. This is unnecessary because the I2S microphone was already initialized during the setup().



```
static bool microphone_inference_start(uint32_t n_samples)
{
    inference.buffer = (int16_t *)malloc(n_samples * sizeof(int16_t));

    if(inference.buffer == NULL) {
        return false;
    }

    inference.buf_count  = 0;
    inference.n_samples  = n_samples;
    inference.buf_ready  = 0;

//    if (i2s_init(EI_CLASSIFIER_FREQUENCY)) {
//        ei_printf("Failed to start I2S!");
//    }

```

Finally, on static void microphone_inference_end(void) function, replace line 243:

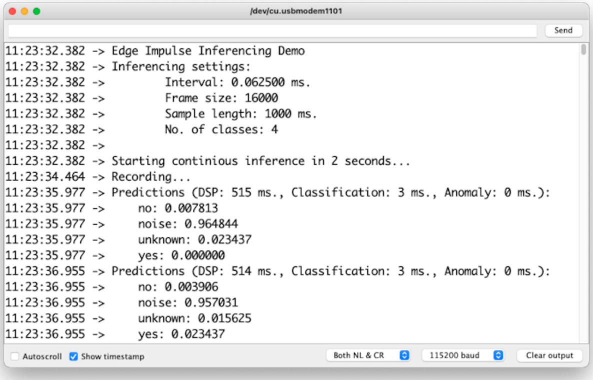


```
241 static void microphone_inference_end(void)
242 {
243     i2s_deinit();
244     ei_free(inference.buffer);
245 }
```

By:

```
static void microphone_inference_end(void)
{
    free(sampleBuffer);
    ei_free(inference.buffer);
}
```

You can find the complete code on the project's GitHub. Upload the sketch to your board and test some real inferences:



```
11:23:32.382 -> Edge Impulse Inferencing Demo
11:23:32.382 -> Inferencing settings:
11:23:32.382 -> 	Interval: 0.062500 ms.
11:23:32.382 -> 	Frame size: 16000
11:23:32.382 -> 	Sample length: 1000 ms.
11:23:32.382 -> 	No. of classes: 4
11:23:32.382 -> 
11:23:32.382 -> Starting continious inference in 2 seconds...
11:23:34.464 -> Recording...
11:23:35.977 -> Predictions (DSP: 515 ms., Classification: 3 ms., Anomaly: 0 ms.):
11:23:35.977 ->     no: 0.007813
11:23:35.977 ->     noise: 0.964844
11:23:35.977 ->     unknown: 0.023437
11:23:35.977 ->     yes: 0.000000
11:23:36.955 -> Predictions (DSP: 514 ms., Classification: 3 ms., Anomaly: 0 ms.):
11:23:36.955 ->     no: 0.003906
11:23:36.955 ->     noise: 0.957031
11:23:36.955 ->     unknown: 0.015625
11:23:36.955 ->     yes: 0.023437
```

Postprocessing

Now that we know the model is working by detecting our keywords, let's modify the code to see the internal LED going on every time a YES is detected.

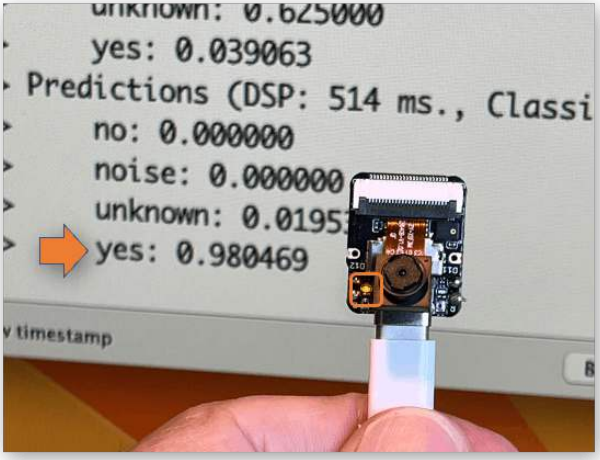
You should initialize the LED:

```
#define LED_BUILT_IN 21
...
void setup()
{
...
  pinMode(LED_BUILT_IN, OUTPUT); // Set the pin as output
  digitalWrite(LED_BUILT_IN, HIGH); //Turn off
...
}
```

And change the // print the predictions portion of the previous code (on loop():

```
int pred_index = 0;     // Initialize pred_index
float pred_value = 0;   // Initialize pred_value

// print the predictions
ei_printf("Predictions ");
ei_printf("(DSP: %d ms., Classification: %d ms., Anomaly: %d ms.)",
     result.timing.dsp, result.timing.classification,
     result.timing.anomaly);
ei_printf(": \n");
for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
      ei_printf("    %s: ", result.classification[ix].label);
      ei_printf_float(result.classification[ix].value);
      ei_printf("\n");

      if (result.classification[ix].value > pred_value){
         pred_index = ix;
         pred_value = result.classification[ix].value;
      }
}

// show the inference result on LED
if (pred_index == 3){
    digitalWrite(LED_BUILT_IN, LOW); //Turn on
}
else{
   digitalWrite(LED_BUILT_IN, HIGH); //Turn off
}
```

You can find the complete code on the project's GitHub. Upload the sketch to your board and test some real inferences:



```
    unknown: 0.625000
    yes: 0.039063
Predictions (DSP: 514 ms., Classi
    no: 0.000000
    noise: 0.000000
    unknown: 0.0195
    yes: 0.980469
```

The idea is that the LED will be ON whenever the keyword YES is detected. In the same way, instead of turning on an LED, this could be a "trigger" for an external device, as we saw in the introduction.

Conclusion

The Seeed XIAO ESP32S3 Sense is a *giant tiny device*! However, it is powerful, trustworthy, not expensive, low power, and has suitable sensors to be used on the most common embedded machine learning applications such as vision and sound. Even though Edge Impulse does not officially support XIAO ESP32S3 Sense (yet!), we realized that using the Studio for training and deployment is straightforward.

> On my GitHub repository, you will find the last version all the codeused on this project and the previous ones of the XIAO ESP32S3 series.

Before we finish, consider that Sound Classification is more than just voice. For example, you can develop TinyML projects around sound in several areas, such as:

- **Security** (Broken Glass detection)
- **Industry** (Anomaly Detection)
- **Medical** (Snore, Toss, Pulmonary diseases)
- **Nature** (Beehive control, insect sound)

Resources

- XIAO ESP32S3 Codes
- Subset of Google Speech Commands Dataset

- KWS MFCC Analysis Colab Notebook
- KWS CNN training Colab Notebook
- XIAO ESP32S3 Post-processing Code
- Edge Impulse Project

Motion Classification and Anomaly Detection

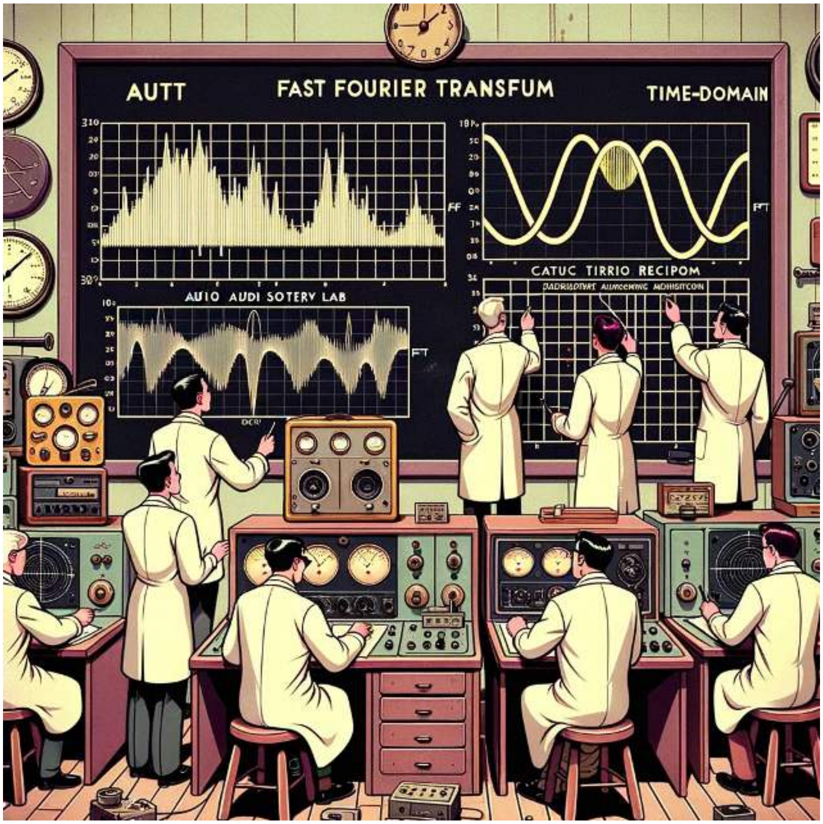
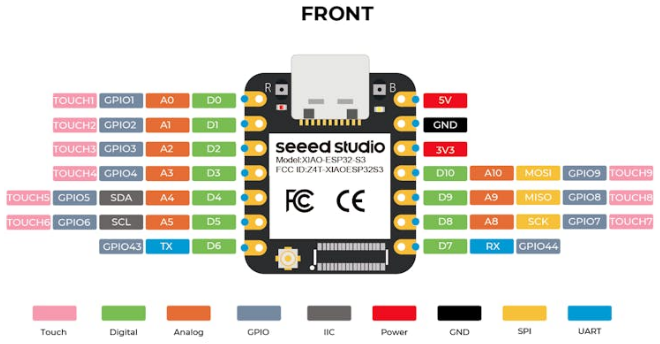


Figure 20.13: *DALL·E prompt - 1950s style cartoon illustration set in a vintage audio lab. Scientists, dressed in classic attire with white lab coats, are intently analyzing audio data on large chalkboards. The boards display intricate FFT (Fast Fourier Transform) graphs and time-domain curves. Antique audio equipment is scattered around, but the data representations are clear and detailed, indicating their focus on audio analysis.*

Overview

The XIAO ESP32S3 Sense, with its built-in camera and mic, is a versatile device. But what if you need to add another type of sensor, such as an IMU? No problem!

One of the standout features of the XIAO ESP32S3 is its multiple pins that can be used as an I2C bus (SDA/SCL pins), making it a suitable platform for sensor integration.



Installing the IMU

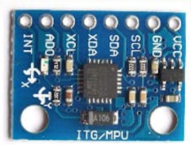
When selecting your IMU, the market offers a wide range of devices, each with unique features and capabilities. You could choose, for example, the ADXL362 (3-axis), MAX21100 (6-axis), MPU6050 (6-axis), LIS3DHTR (3-axis), or the LCM20600Seeed Grove— (6-axis), which is part of the IMU 9DOF (lcm20600+AK09918). This variety allows you to tailor your choice to your project's specific needs.

For this project, we will use an IMU, the MPU6050 (or 6500), a low-cost (less than 2.00 USD) 6-axis Accelerometer/Gyroscope unit.

At the end of the lab, we will also comment on using the LCM20600.

The MPU-6500 is a 6-axis Motion Tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, and a Digital Motion ProcessorTM (DMP) in a small 3x3x0.9mm package. It also features a 4096-byte FIFO that can lower the traffic on the serial bus interface and reduce power consumption by allowing the system processor to burst read sensor data and then go into a low-power mode.

With its dedicated I2C sensor bus, the MPU-6500 directly accepts inputs from external I2C devices. MPU-6500, with its 6-axis integration, on-chip DMP, and run-time calibration firmware, enables manufacturers to eliminate the costly and complex selection, qualification, and system-level integration of discrete devices, guaranteeing optimal motion performance for consumers. MPU-6500 is also designed to interface with multiple non-inertial digital sensors, such as pressure sensors, on its auxiliary I2C port.



MPU6050

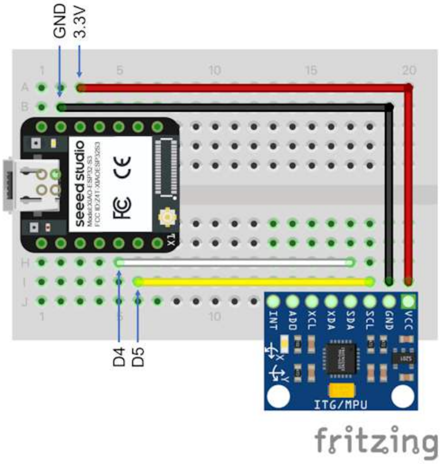
MPU6500

Usually, the libraries available are for MPU6050, but they work for both devices.

Connecting the HW

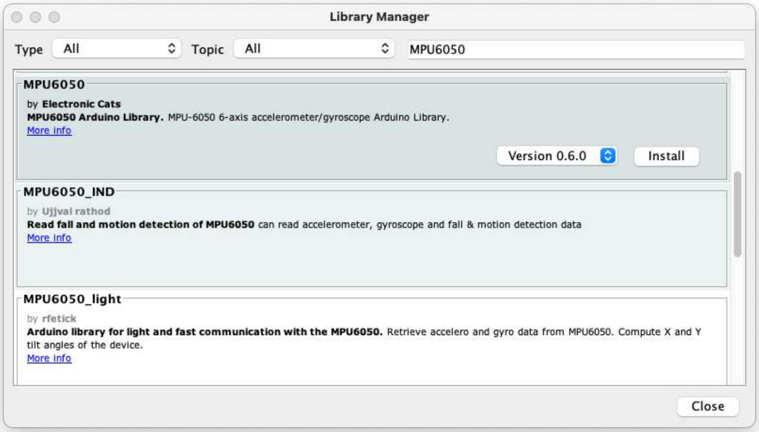
Connect the IMU to the XIAO according to the below diagram:

- MPU6050 **SCL** –> XIAO **D5**
- MPU6050 **SDA** –> XIAO **D4**
- MPU6050 **VCC** –> XIAO **3.3V**
- MPU6050 **GND** –> XIAO **GND**



Install the Library

Go to Arduino Library Manager and type MPU6050. Install the latest version.



Library Manager

Type All Topic All MPU6050

MPU6050
by **Electronic Cats**
MPU6050 Arduino Library. MPU-6050 6-axis accelerometer/gyroscope Arduino Library.
More info
Version 0.6.0 Install

MPU6050_IND
by Ujjval rathod
Read fall and motion detection of MPU6050 can read accelerometer, gyroscope and fall & motion detection data
More info

MPU6050_light
by rfetick
Arduino library for light and fast communication with the MPU6050. Retrieve accelero and gyro data from MPU6050. Compute X and Y tilt angles of the device.
More info

Close

Download the sketch MPU6050_Acc_Data_Acquisition.in:

```
/*
 * Based on I2C device class (I2Cdev) Arduino sketch for MPU6050 class
   by Jeff Rowberg <jeff@rowberg.net>
 * and Edge Impulse Data Forwarder Exampe (Arduino)
   - https://docs.edgeimpulse.com/docs/cli-data-forwarder
 *
 * Developed by M.Rovai @11May23
 */

#include "I2Cdev.h"
#include "MPU6050.h"
#include "Wire.h"

#define FREQUENCY_HZ        50
#define INTERVAL_MS         (1000 / (FREQUENCY_HZ + 1))
#define ACC_RANGE           1 // 0: -/+2G; 1: +/-4G

// convert factor g to m/s^2^ ==> [-32768, +32767] ==> [-2g, +2g]
#define CONVERT_G_TO_MS2    (9.81/(16384.0/(1.+ACC_RANGE)))

static unsigned long last_interval_ms = 0;

MPU6050 imu;
int16_t ax, ay, az;

void setup() {

    Serial.begin(115200);


    // initialize device
    Serial.println("Initializing I2C devices...");
    Wire.begin();
    imu.initialize();
    delay(10);

//    // verify connection
//    if (imu.testConnection()) {
//      Serial.println("IMU connected");
//    }
//    else {
//      Serial.println("IMU Error");
//    }
    delay(300);

    //Set MCU 6050 OffSet Calibration
    imu.setXAccelOffset(-4732);
    imu.setYAccelOffset(4703);
    imu.setZAccelOffset(8867);
    imu.setXGyroOffset(61);
    imu.setYGyroOffset(-73);
    imu.setZGyroOffset(35);

    /* Set full-scale accelerometer range.
     * 0 = +/- 2g
     * 1 = +/- 4g
     * 2 = +/- 8g
     * 3 = +/- 16g
     */
    imu.setFullScaleAccelRange(ACC_RANGE);
}

void loop() {

      if (millis() > last_interval_ms + INTERVAL_MS) {
        last_interval_ms = millis();

        // read raw accel/gyro measurements from device
        imu.getAcceleration(&ax, &ay, &az);

        // converting to m/s^2^
        float ax_m_s^2^ = ax * CONVERT_G_TO_MS2;
        float ay_m_s^2^ = ay * CONVERT_G_TO_MS2;
        float az_m_s^2^ = az * CONVERT_G_TO_MS2;

        Serial.print(ax_m_s^2^);
        Serial.print("\t");
        Serial.print(ay_m_s^2^);
        Serial.print("\t");
        Serial.println(az_m_s^2^);
      }
}
```

Some comments about the code:

Note that the values generated by the accelerometer and gyroscope have a range: [-32768, +32767], so for example, if the default accelerometer range is used, the range in Gs should be: [-2g, +2g]. So, "1G" means 16384.

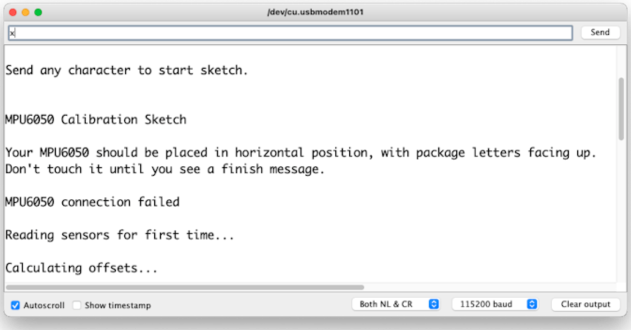
For conversion to m/s^2, for example, you can define the following:

```
#define CONVERT_G_TO_MS2 (9.81/16384.0)
```

In the code, I left an option (ACC_RANGE) to be set to 0 (+/-2G) or 1 (+/-4G). We will use +/-4G; that should be enough for us. In this case.

We will capture the accelerometer data on a frequency of 50Hz, and the acceleration data will be sent to the Serial Port as meters per squared second (m/s^2).

When you ran the code with the IMU resting over your table, the accelerometer data shown on the Serial Monitor should be around 0.00, 0.00, and 9.81. If the values are a lot different, you should calibrate the IMU.
The MCU6050 can be calibrated using the sketch: mcu6050-calibration.ino.
Run the code. The following will be displayed on the Serial Monitor:

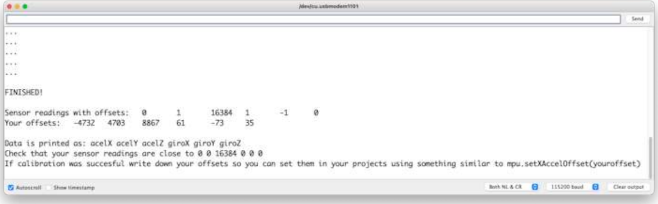


```
Send any character to start sketch.


MPU6050 Calibration Sketch

Your MPU6050 should be placed in horizontal position, with package letters facing up.
Don't touch it until you see a finish message.

MPU6050 connection failed

Reading sensors for first time...

Calculating offsets...
```

Send any character (in the above example, “x”), and the calibration should start.

> Note that a message MPU6050 connection failed. Ignore this message. For some reason, imu.testConnection() is not returning a correct result.

In the end, you will receive the offset values to be used on all your sketches:

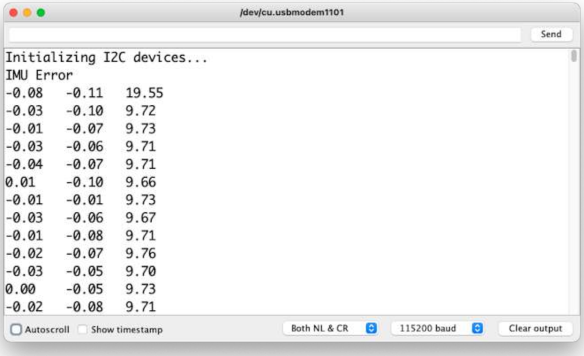


```
...
...
...
...
...

FINISHED!

Sensor readings with offsets:   0       1       16384   1       -1      0
Your offsets:   -4732   4703    8867    61      -73     35

Data is printed as: acelX acelY acelZ giroX giroY giroZ
Check that your sensor readings are close to 0 0 16384 0 0 0
If calibration was succesful write down your offsets so you can set them in your projects using something similar to mpu.setXAccelOffset(youroffset)
```

Take the values and use them on the setup:

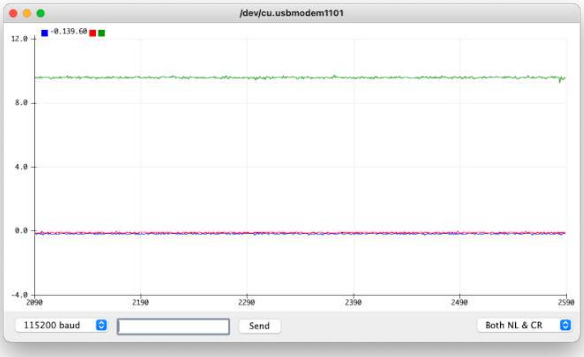
```
//Set MCU 6050 OffSet Calibration
imu.setXAccelOffset(-4732);
imu.setYAccelOffset(4703);
imu.setZAccelOffset(8867);
imu.setXGyroOffset(61);
imu.setYGyroOffset(-73);
imu.setZGyroOffset(35);
```

Now, run the sketch MPU6050_Acc_Data_Acquisition.in:
Once you run the above sketch, open the Serial Monitor:

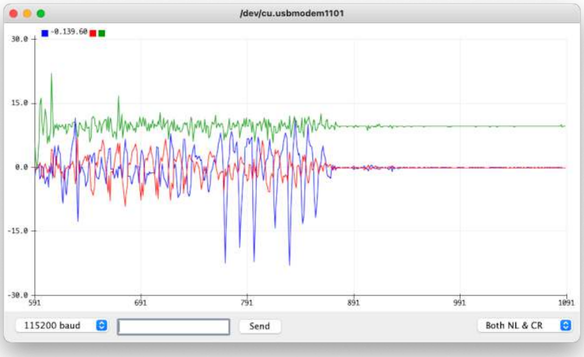


```
Initializing I2C devices...
IMU Error
-0.08   -0.11   19.55
-0.03   -0.10   9.72
-0.01   -0.07   9.73
-0.03   -0.06   9.71
-0.04   -0.07   9.71
0.01    -0.10   9.66
-0.01   -0.01   9.73
-0.03   -0.06   9.67
-0.01   -0.08   9.71
-0.02   -0.07   9.76
-0.03   -0.05   9.70
0.00    -0.05   9.73
-0.02   -0.08   9.71
```

Or check the Plotter:



Move your device in the three axes. You should see the variation on Plotter:

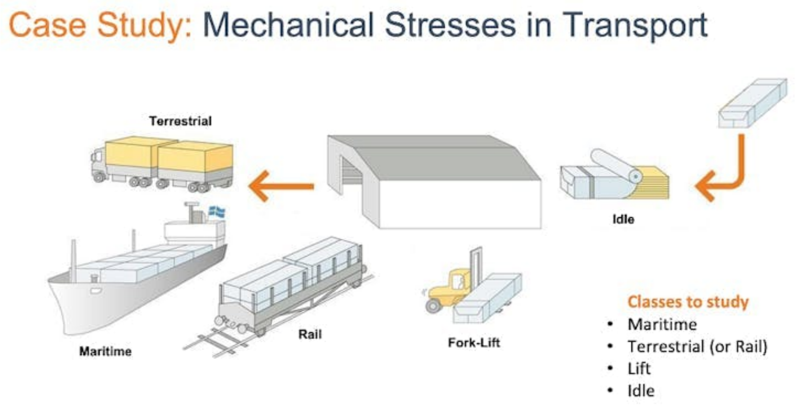


The TinyML Motion Classification Project

For our lab, we will simulate mechanical stresses in transport. Our problem will be to classify four classes of movement:

- **Maritime** (pallets in boats)
- **Terrestrial** (palettes in a Truck or Train)
- **Lift** (Palettes being handled by Fork-Lift)
- **Idle** (Palettes in Storage houses)

So, to start, we should collect data. Then, accelerometers will provide the data on the palette (or container).



Case Study: Mechanical Stresses in Transport

Classes to study

- Maritime
- Terrestrial (or Rail)
- Lift
- Idle

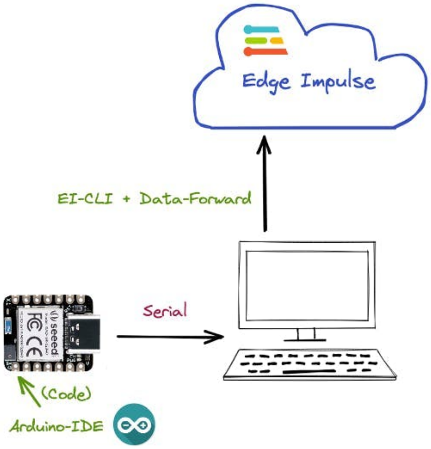
From the above images, we can see that primarily horizontal movements should be associated with the "Terrestrial class," Vertical movements with the "Lift Class," no activity with the "Idle class," and movement on all three axes to Maritime class.

Connecting the device to Edge Impulse

For data collection, we should first connect our device to the Edge Impulse Studio, which will also be used for data pre-processing, model training, testing, and deployment.

> Follow the instructions hereto install the Node.jsand Edge Impulse CLI on your computer.

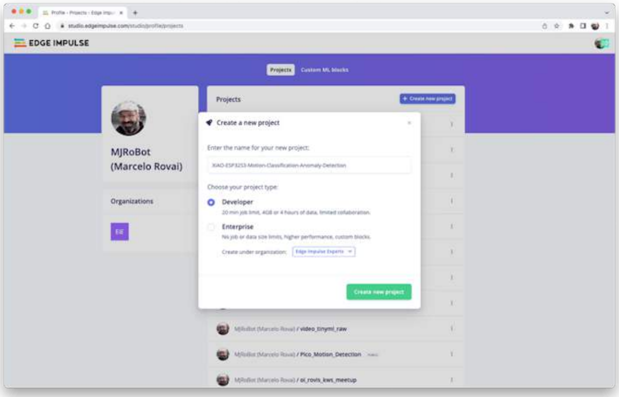
Once the XIAO ESP32S3 is not a fully supported development board by Edge Impulse, we should, for example, use the CLI Data Forwarder to capture data from our sensor and send it to the Studio, as shown in this diagram:



> You can alternately capture your data "offline," store them on an SD card or send them to your computer via Bluetooth or Wi-Fi. In this video, you can learn alternative ways to send data to the Edge Impulse Studio.

Connect your device to the serial port and run the previous code to capture IMU (Accelerometer) data, "printing them" on the serial. This will allow the Edge Impulse Studio to "capture" them.

Go to the Edge Impulse page and create a project.



Create a new project

Enter the name for your new project:

XIAO-ESP32S3-Motion-Classification-Anomaly-Detection

Choose your project type:

● Developer
20 min job limit, 4GB or 4 hours of data, limited collaboration.

○ Enterprise
No job or data size limits, higher performance, custom blocks.
Create under organization: Edge Impulse Experts

Create new project

MJRoBot (Marcelo Rovai) / video_tinyml_raw

MJRoBot (Marcelo Rovai) / Pico_Motion_Detection

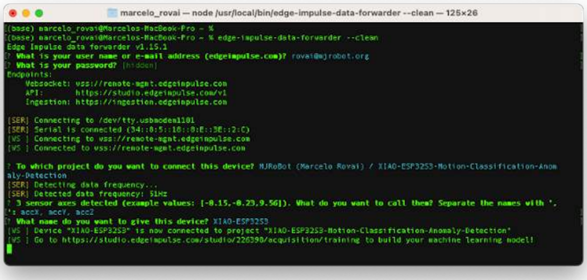
MJRoBot (Marcelo Rovai) / ol_rovis_kws_meetup

> The maximum length for an Arduino library name is **63 characters**. Note that the Studio will name the final library using your project name and include "_inference" to it. The name I chose initially did not work when I tried to deploy the Arduino library because it resulted in 64 characters. So, I need to change it by taking out the "anomaly detection" part.

Start the CLI Data Forwarderon your terminal, entering (if it is the first time) the following command:

```
edge-impulse-data-forwarder --clean
```

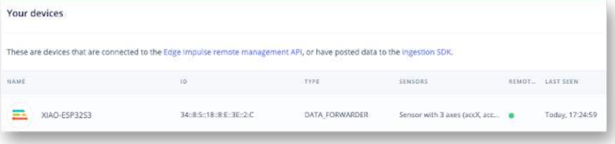
Next, enter your EI credentials and choose your project, variables, and device names:



marcelo_rovai — node /usr/local/bin/edge-impulse-data-forwarder --clean — 125×26

```
(base) marcelo_rovai@Marcelos-MacBook-Pro ~ %
(base) marcelo_rovai@Marcelos-MacBook-Pro ~ % edge-impulse-data-forwarder --clean
Edge Impulse data forwarder v1.15.1
? What is your user name or e-mail address (edgeimpulse.com)? rovai@mjrobot.org
? What is your password? [hidden]
Endpoints:
    Websocket: wss://remote-mgmt.edgeimpulse.com
    API:       https://studio.edgeimpulse.com/v1
    Ingestion: https://ingestion.edgeimpulse.com

[SER] Connecting to /dev/tty.usbmodem1101
[SER] Serial is connected (34::8:5::18::8:E::3E::2:C)
[WS ] Connecting to wss://remote-mgmt.edgeimpulse.com
[WS ] Connected to wss://remote-mgmt.edgeimpulse.com

? To which project do you want to connect this device? MJRoBot (Marcelo Rovai) / XIAO-ESP32S3-Motion-Classification-Anom
aly-Detection
[SER] Detecting data frequency...
[SER] Detected data frequency: 51Hz
? 3 sensor axes detected (example values: [-0.15,-0.23,9.56]). What do you want to call them? Separate the names with ',
': accX, accY, accZ
? What name do you want to give this device? XIAO-ESP32S3
[WS ] Device "XIAO-ESP32S3" is now connected to project "XIAO-ESP32S3-Motion-Classification-Anomaly-Detection"
[WS ] Go to https://studio.edgeimpulse.com/studio/226398/acquisition/training to build your machine learning model!
```

Go to your EI Project and verify if the device is connected (the dot should be green):



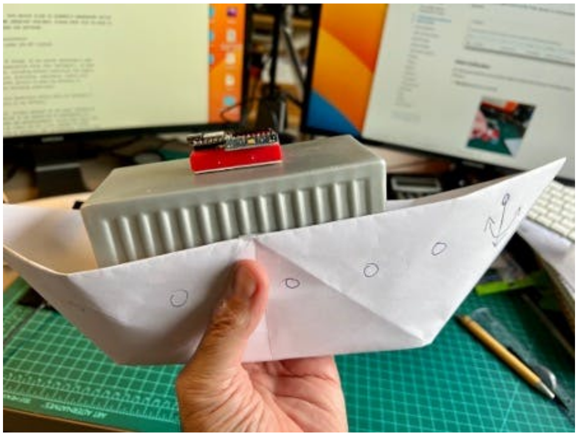
Your devices

These are devices that are connected to the Edge Impulse remote management API, or have posted data to the ingestion SDK.

NAME	ID	TYPE	SENSORS	REMOT...	LAST SEEN
XIAO-ESP32S3	34::8:5::18::8:E::3E::2:C	DATA_FORWARDER	Sensor with 3 axes (accX, acc...	●	Today, 17:24:59

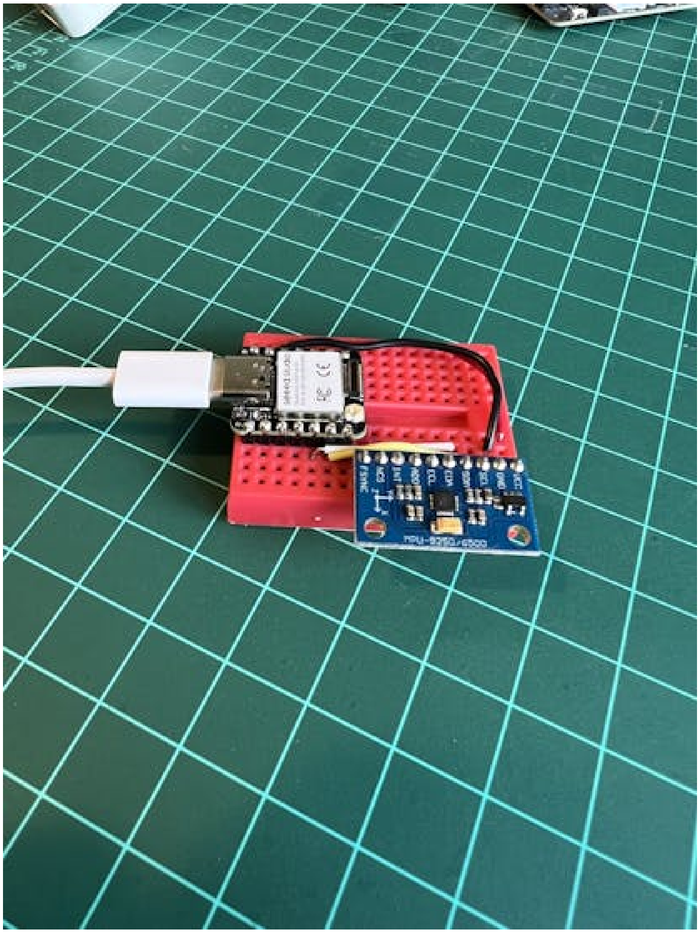
Data Collection

As discussed before, we should capture data from all four Transportation Classes. Imagine that you have a container with a built-in accelerometer:

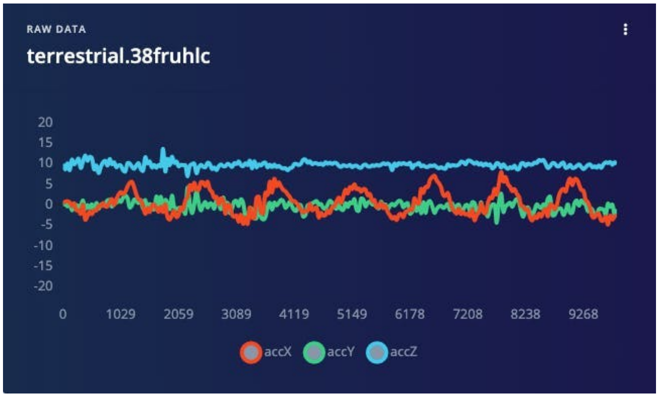


Now imagine your container is on a boat, facing an angry ocean, on a truck, etc.:

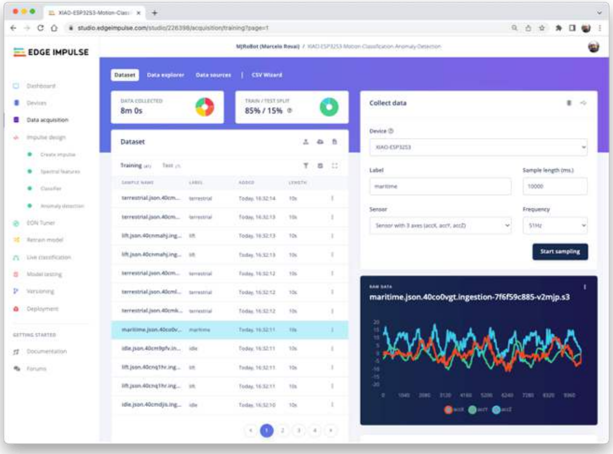
- **Maritime** (pallets in boats)
  - Move the XIAO in all directions, simulating an undulatory boat movement.
- **Terrestrial** (palettes in a Truck or Train)
  - Move the XIAO over a horizontal line.
- **Lift** (Palettes being handled by Fork-Lift)
  - Move the XIAO over a vertical line.
- **Idle** (Palettes in Storage houses)
  - Leave the XIAO over the table.



Below is one sample (raw data) of 10 seconds:



You can capture, for example, 2 minutes (twelve samples of 10 seconds each) for the four classes. Using the "3 dots" after each one of the samples, select 2, moving them for the Test set (or use the automatic Train/Test Split tool on the Danger Zone of Dashboard tab). Below, you can see the result datasets:



DATA COLLECTED 8m 0s

TRAIN / TEST SPLIT 85% / 15%

Dataset

SAMPLE NAME	LABEL	ADDED	LENGTH
terrestrial.json.40cm...	terrestrial	Today, 16:32:14	10s
terrestrial.json.40cm...	terrestrial	Today, 16:32:13	10s
lift.json.40cnmahj.ing...	lift	Today, 16:32:13	10s
lift.json.40cnmahj.ing...	lift	Today, 16:32:13	10s
terrestrial.json.40cm...	terrestrial	Today, 16:32:12	10s
terrestrial.json.40cml...	terrestrial	Today, 16:32:12	10s
terrestrial.json.40cmk...	terrestrial	Today, 16:32:12	10s
maritime.json.40co0v...	maritime	Today, 16:32:11	10s
idle.json.40cm9pfv.in...	idle	Today, 16:32:11	10s
lift.json.40cnq1hr.ing...	lift	Today, 16:32:11	10s
lift.json.40cnq1hr.ing...	lift	Today, 16:32:11	10s
idle.json.40cmdjs.ing...	idle	Today, 16:32:10	10s

Collect data

Device: XIAO-ESP32S3

Label: maritime

Sample length (ms.): 10000

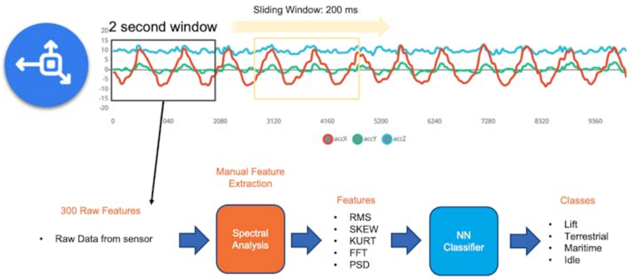
Sensor: Sensor with 3 axes (accX, accY, accZ)

Frequency: 51Hz

RAW DATA
maritime.json.40co0vgt.ingestion-7f6f59c885-v2mjp.s3

Data Pre-Processing

The raw data type captured by the accelerometer is a "time series" and should be converted to "tabular data". We can do this conversion using a sliding window over the sample data. For example, in the below figure,



We can see 10 seconds of accelerometer data captured with a sample rate (SR) of 50 Hz. A 2-second window will capture 300 data points (3 axis × 2 seconds × 50 samples). We will slide this window each 200ms, creating a larger dataset where each instance has 300 raw features.

> You should use the best SR for your case, considering Nyquist's theorem, which states that a periodic signal must be sampled at more than twice the signal's highest frequency component.

Data preprocessing is a challenging area for embedded machine learning. Still, Edge Impulse helps overcome this with its digital signal processing (DSP) preprocessing step and, more specifically, the Spectral Features.

On the Studio, this dataset will be the input of a Spectral Analysis block, which is excellent for analyzing repetitive motion, such as data from accelerometers. This block will perform a DSP (Digital Signal Processing), extracting features such as "FFT" or "Wavelets". In the most common case, FFT, the **Time Domain Statistical features** per axis/channel are:

- RMS
- Skewness
- Kurtosis

And the **Frequency Domain Spectral features** per axis/channel are:

- Spectral Power
- Skewness
- Kurtosis

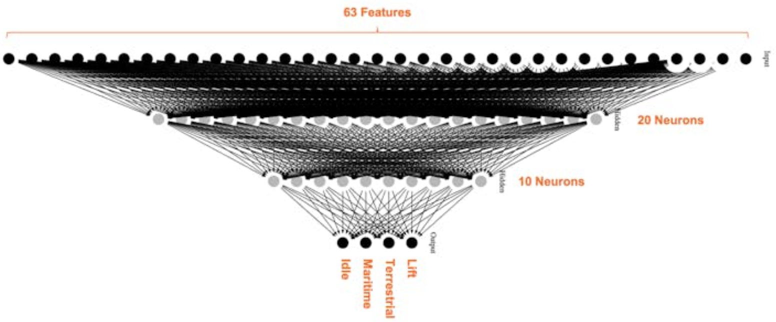
For example, for an FFT length of 32 points, the Spectral Analysis Block's resulting output will be 21 features per axis (a total of 63 features).

Those 63 features will be the Input Tensor of a Neural Network Classifier and the Anomaly Detection model (K-Means).

> You can learn more by digging into the lab DSP Spectral Features

Model Design

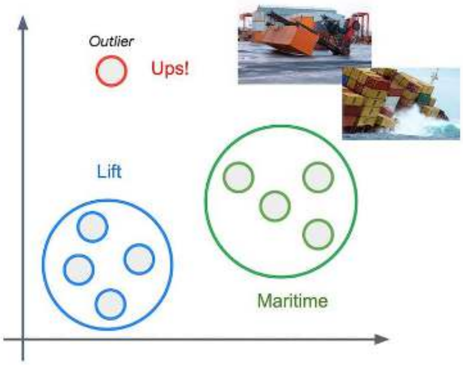
Our classifier will be a Dense Neural Network (DNN) that will have 63 neurons on its input layer, two hidden layers with 20 and 10 neurons, and an output layer with four neurons (one per each class), as shown here:



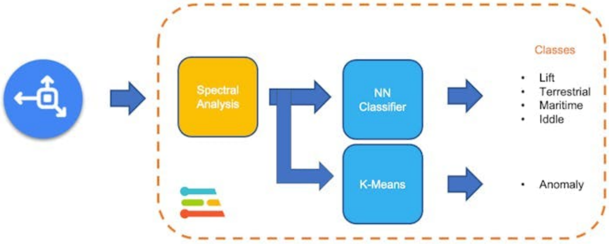
Impulse Design

An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

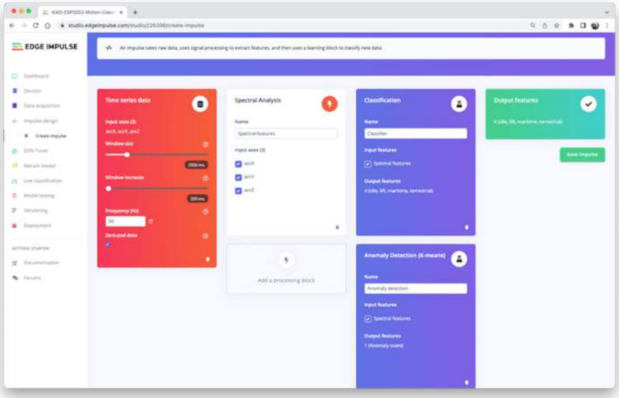
We also take advantage of a second model, the K-means, that can be used for Anomaly Detection. If we imagine that we could have our known classes as clusters, any sample that could not fit on that could be an outlier, an anomaly (for example, a container rolling out of a ship on the ocean).



Imagine our XIAO rolling or moving upside-down, on a movement complement different from the one trained

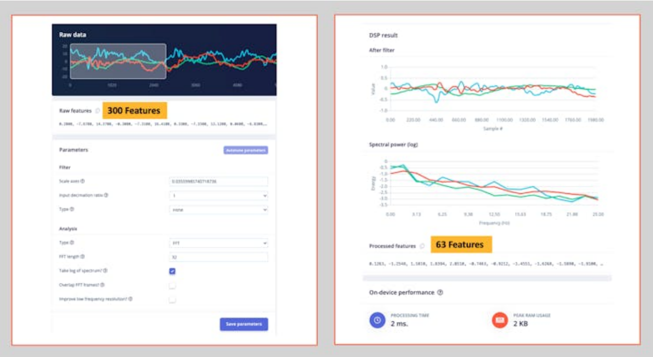


Below is our final Impulse design:



Generating features

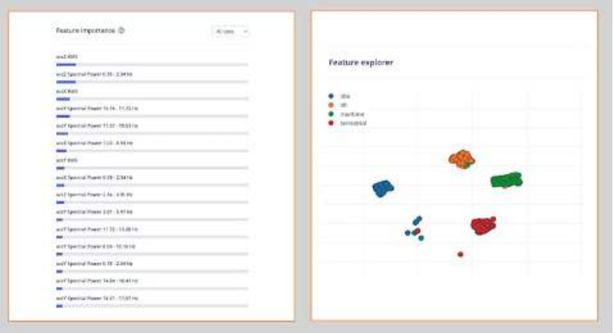
At this point in our project, we have defined the pre-processing method and the model designed. Now, it is time to have the job done. First, let's take the raw data (time-series type) and convert it to tabular data. Go to the Spectral Features tab and select Save Parameters:



At the top menu, select the Generate Features option and the Generate Features button. Each 2-second window data will be converted into one data point of 63 features.

> The Feature Explorer will show those data in 2D using UMAP. Uniform Manifold Approximation and Projection (UMAP) is a dimension reduction technique that can be used for visualization similarly to t-SNE but also for general non-linear dimension reduction.

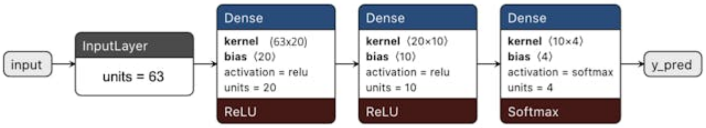
The visualization allows one to verify that the classes present an excellent separation, which indicates that the classifier should work well.

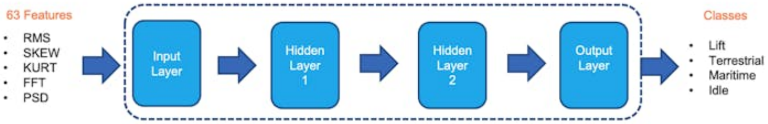


Optionally, you can analyze the relative importance of each feature for one class compared with other classes.

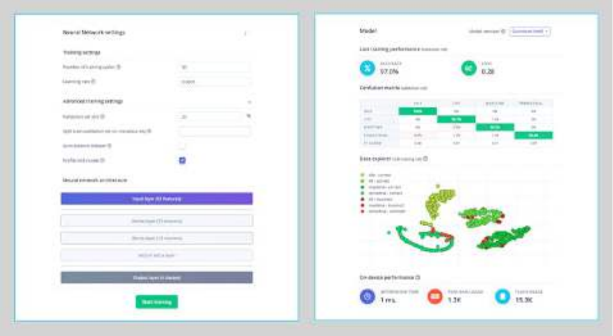
Training

Our model has four layers, as shown below:

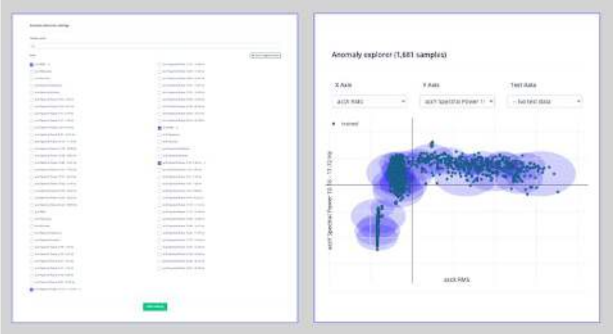




As hyperparameters, we will use a Learning Rate of 0.005 and 20% of data for validation for 30 epochs. After training, we can see that the accuracy is 97%.

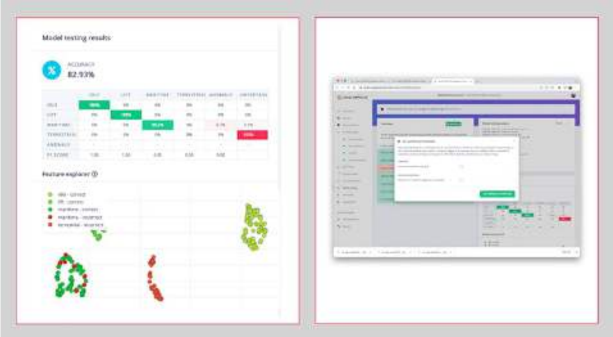


For anomaly detection, we should choose the suggested features that are precisely the most important in feature extraction. The number of clusters will be 32, as suggested by the Studio:

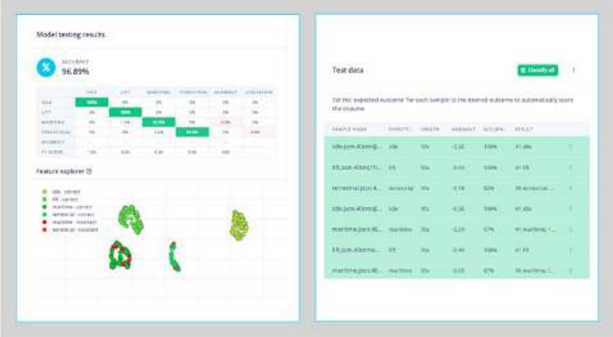


Testing

Using 20% of the data left behind during the data capture phase, we can verify how our model will behave with unknown data; if not 100% (what is expected), the result was not that good (8%), mainly due to the terrestrial class. Once we have four classes (which output should add 1.0), we can set up a lower threshold for a class to be considered valid (for example, 0.4):



Now, the Test accuracy will go up to 97%.

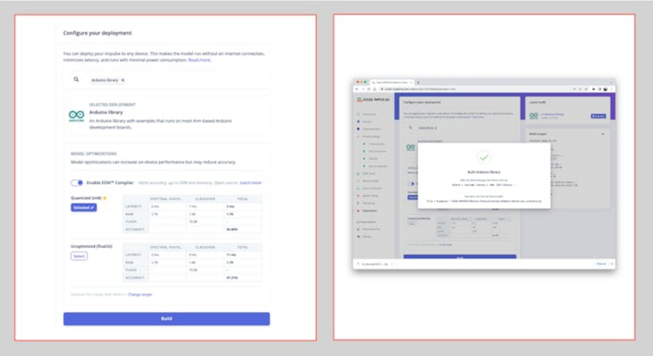


You should also use your device (which is still connected to the Studio) and perform some Live Classification.

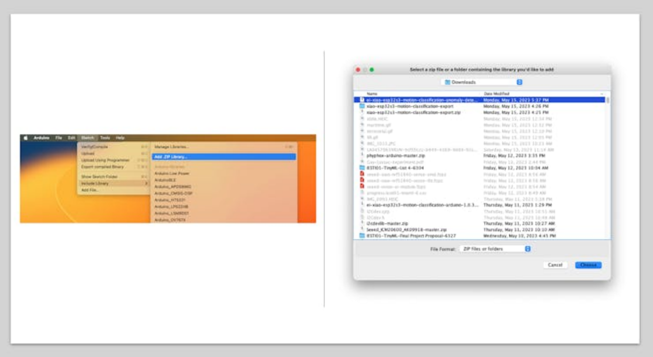
> Be aware that here you will capture real data with your device and upload it to the Studio, where an inference will be taken using the trained model (But the model is NOT in your device).

Deploy

Now it is time for magic! The Studio will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. You should select the option Arduino Library, and at the bottom, choose Quantized (Int8) and Build. A Zip file will be created and downloaded to your computer.



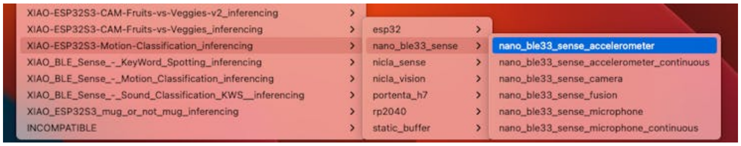
On your Arduino IDE, go to the Sketch tab, select the option Add.ZIP Library, and Choose the.zip file downloaded by the Studio:



Inference

Now, it is time for a real test. We will make inferences that are wholly disconnected from the Studio. Let's change one of the code examples created when you deploy the Arduino Library.

In your Arduino IDE, go to the File/Examples tab and look for your project, and on examples, select nano_ble_sense_accelerometer:


XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v2_inferencing
XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing
XIAO-ESP32S3-Motion-Classification_inferencing
XIAO_BLE_Sense_-_KeyWord_Spotting_inferencing
XIAO_BLE_Sense_-_Motion_Classification_inferencing
XIAO_BLE_Sense_-_Sound_Classification_KWS__inferencing
XIAO_ESP32S3_mug_or_not_mug_inferencing
INCOMPATIBLE
esp32
nano_ble33_sense
nicla_sense
nicla_vision
portenta_h7
rp2040
static_buffer
nano_ble33_sense_accelerometer
nano_ble33_sense_accelerometer_continuous
nano_ble33_sense_camera
nano_ble33_sense_fusion
nano_ble33_sense_microphone
nano_ble33_sense_microphone_continuous

Of course, this is not your board, but we can have the code working with only a few changes.

For example, at the beginning of the code, you have the library related to Arduino Sense IMU:

```
/* Includes --------------------------------------------- */
#include <XIAO-ESP32S3-Motion-Classification_inferencing.h>
#include <Arduino_LSM9DS1.h>
```

Change the "includes" portion with the code related to the IMU:

```
#include <XIAO-ESP32S3-Motion-Classification_inferencing.h>
#include "I2Cdev.h"
#include "MPU6050.h"
#include "Wire.h"
```

Change the Constant Defines

```
/* Constant defines ---------------------------------- */
MPU6050 imu;
int16_t ax, ay, az;

#define ACC_RANGE           1 // 0: -/+2G; 1: +/-4G
#define CONVERT_G_TO_MS2    (9.81/(16384/(1.+ACC_RANGE)))
#define MAX_ACCEPTED_RANGE  (2*9.81)+(2*9.81)*ACC_RANGE
```

On the setup function, initiate the IMU set the off-set values and range:

```
// initialize device
Serial.println("Initializing I2C devices...");
Wire.begin();
imu.initialize();
delay(10);

//Set MCU 6050 OffSet Calibration
imu.setXAccelOffset(-4732);
imu.setYAccelOffset(4703);
imu.setZAccelOffset(8867);
imu.setXGyroOffset(61);
imu.setYGyroOffset(-73);
imu.setZGyroOffset(35);

imu.setFullScaleAccelRange(ACC_RANGE);
```

At the loop function, the buffers buffer[ix], buffer[ix + 1], and buffer[ix + 2] will receive the 3-axis data captured by the accelerometer. On the original code, you have the line:

```
IMU.readAcceleration(buffer[ix], buffer[ix + 1], buffer[ix + 2]);
```

Change it with this block of code:

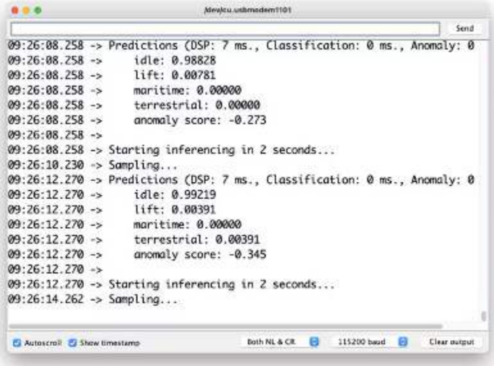
```
imu.getAcceleration(&ax, &ay, &az);
buffer[ix + 0] = ax;
buffer[ix + 1] = ay;
buffer[ix + 2] = az;
```

You should change the order of the following two blocks of code. First, you make the conversion to raw data to “Meters per squared second (ms^2)”, followed by the test regarding the maximum acceptance range (that here is in ms^2, but on Arduino, was in Gs):

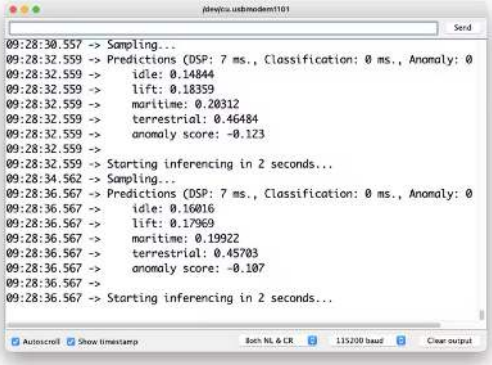
```
buffer[ix + 0] *= CONVERT_G_TO_MS2;
buffer[ix + 1] *= CONVERT_G_TO_MS2;
buffer[ix + 2] *= CONVERT_G_TO_MS2;

for (int i = 0; i < 3; i++) {
     if (fabs(buffer[ix + i]) > MAX_ACCEPTED_RANGE) {
        buffer[ix + i] = ei_get_sign(buffer[ix + i])
                              * MAX_ACCEPTED_RANGE;
     }
}
```

And that is it! You can now upload the code to your device and proceed with the inferences. The complete code is available on the project’s GitHub.

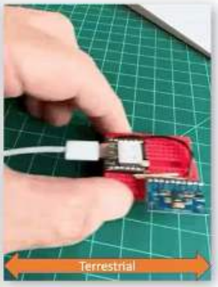
Now you should try your movements, seeing the result of the inference of each class on the images:

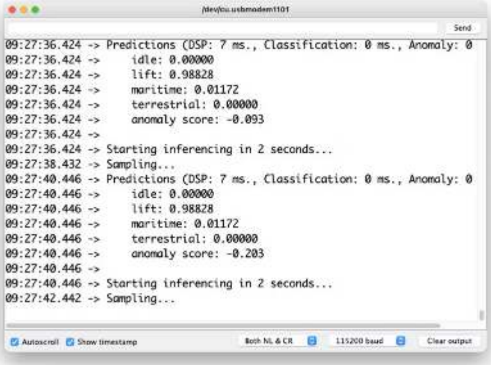


```
09:26:08.258 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:26:08.258 ->     idle: 0.98828
09:26:08.258 ->     lift: 0.00781
09:26:08.258 ->     maritime: 0.00000
09:26:08.258 ->     terrestrial: 0.00000
09:26:08.258 ->     anomaly score: -0.273
09:26:08.258 ->
09:26:08.258 -> Starting inferencing in 2 seconds...
09:26:10.230 -> Sampling...
09:26:12.270 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:26:12.270 ->     idle: 0.99219
09:26:12.270 ->     lift: 0.00391
09:26:12.270 ->     maritime: 0.00000
09:26:12.270 ->     terrestrial: 0.00391
09:26:12.270 ->     anomaly score: -0.345
09:26:12.270 ->
09:26:12.270 -> Starting inferencing in 2 seconds...
09:26:14.262 -> Sampling...
```

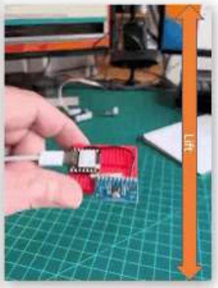


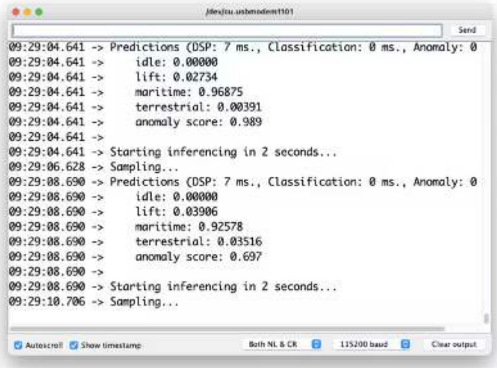
```
09:28:30.557 -> Sampling...
09:28:32.559 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:28:32.559 ->     idle: 0.14844
09:28:32.559 ->     lift: 0.18359
09:28:32.559 ->     maritime: 0.20312
09:28:32.559 ->     terrestrial: 0.46484
09:28:32.559 ->     anomaly score: -0.123
09:28:32.559 ->
09:28:32.559 -> Starting inferencing in 2 seconds...
09:28:34.562 -> Sampling...
09:28:36.567 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:28:36.567 ->     idle: 0.16016
09:28:36.567 ->     lift: 0.17969
09:28:36.567 ->     maritime: 0.19922
09:28:36.567 ->     terrestrial: 0.45703
09:28:36.567 ->     anomaly score: -0.107
09:28:36.567 ->
09:28:36.567 -> Starting inferencing in 2 seconds...
```



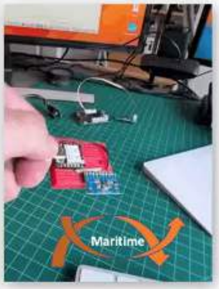


```
09:27:36.424 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:27:36.424 ->     idle: 0.00000
09:27:36.424 ->     lift: 0.98828
09:27:36.424 ->     maritime: 0.01172
09:27:36.424 ->     terrestrial: 0.00000
09:27:36.424 ->     anomaly score: -0.093
09:27:36.424 ->
09:27:36.424 -> Starting inferencing in 2 seconds...
09:27:38.432 -> Sampling...
09:27:40.446 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:27:40.446 ->     idle: 0.00000
09:27:40.446 ->     lift: 0.98828
09:27:40.446 ->     maritime: 0.01172
09:27:40.446 ->     terrestrial: 0.00000
09:27:40.446 ->     anomaly score: -0.203
09:27:40.446 ->
09:27:40.446 -> Starting inferencing in 2 seconds...
09:27:42.442 -> Sampling...
```

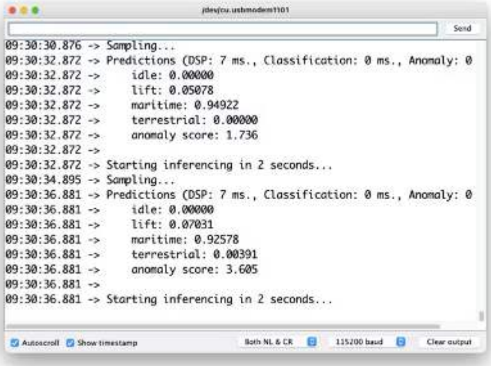




```
09:29:04.641 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:29:04.641 ->     idle: 0.00000
09:29:04.641 ->     lift: 0.02734
09:29:04.641 ->     maritime: 0.96875
09:29:04.641 ->     terrestrial: 0.00391
09:29:04.641 ->     anomaly score: 0.989
09:29:04.641 ->
09:29:04.641 -> Starting inferencing in 2 seconds...
09:29:06.628 -> Sampling...
09:29:08.690 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:29:08.690 ->     idle: 0.00000
09:29:08.690 ->     lift: 0.03906
09:29:08.690 ->     maritime: 0.92578
09:29:08.690 ->     terrestrial: 0.03516
09:29:08.690 ->     anomaly score: 0.697
09:29:08.690 ->
09:29:08.690 -> Starting inferencing in 2 seconds...
09:29:10.706 -> Sampling...
```



And, of course, some "anomaly", for example, putting the XIAO upside-down. The anomaly score will be over 1:

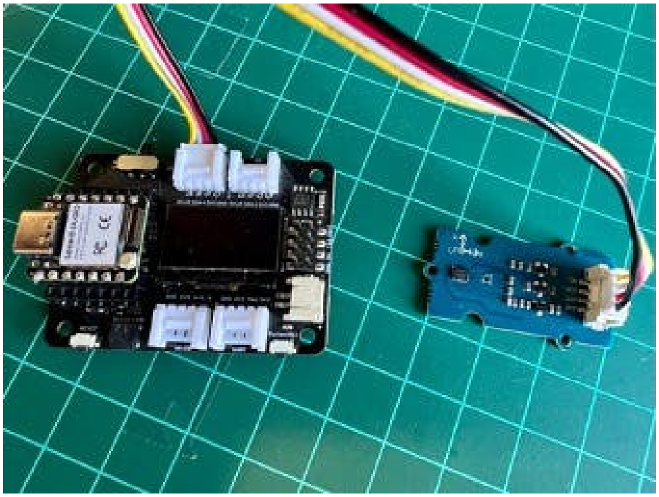


```
09:30:30.876 -> Sampling...
09:30:32.872 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:30:32.872 ->     idle: 0.00000
09:30:32.872 ->     lift: 0.05078
09:30:32.872 ->     maritime: 0.94922
09:30:32.872 ->     terrestrial: 0.00000
09:30:32.872 ->     anomaly score: 1.736
09:30:32.872 ->
09:30:32.872 -> Starting inferencing in 2 seconds...
09:30:34.895 -> Sampling...
09:30:36.881 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:30:36.881 ->     idle: 0.00000
09:30:36.881 ->     lift: 0.07031
09:30:36.881 ->     maritime: 0.92578
09:30:36.881 ->     terrestrial: 0.00391
09:30:36.881 ->     anomaly score: 3.605
09:30:36.881 ->
09:30:36.881 -> Starting inferencing in 2 seconds...
```

☑ Autoscroll ☑ Show timestamp Both NL & CR 115200 baud Clear output

Conclusion

Regarding the IMU, this project used the low-cost MPU6050 but could also use other IMUs, for example, the LCM20600 (6-axis), which is part of the Seeed Grove - IMU 9DOF (lcm20600+AK09918). You can take advantage of this sensor, which has integrated a Grove connector, which can be helpful in the case you use the XIAO with an extension board, as shown below:



You can follow the instructions here to connect the IMU with the MCU. Only note that for using the Grove ICM20600 Accelerometer, it is essential to update the files **I2Cdev.cpp** and **I2Cdev.h** that you will download from the library provided by Seeed Studio. For that, replace both files from this link. You can find a sketch for testing the IMU on the GitHub project: accelerometer_test.ino.

> On the projet's GitHub repository, you will find the last version of all codeand other docs: XIAO-ESP32S3 - IMU.

Resources

- XIAO ESP32S3 Codes
- Edge Impulse Spectral Features Block Colab Notebook
- Edge Impulse Project

Grove Vision AI V2

These labs offer an opportunity to gain practical experience with machine learning (ML) systems on a high-end, yet compact, embedded device, the Seeed Studio Grove Vision AI V2. Unlike working with large models requiring data center-scale resources, these labs allow you to interact with hardware and software using TinyML directly. This hands-on approach provides a tangible understanding of the challenges and opportunities in deploying AI, albeit on a small scale. However, the principles are essentially the same as what you would encounter when working with larger or even smaller systems.

The Grove Vision AI V2 occupies a unique position in the embedded AI landscape, bridging the gap between basic microcontroller solutions, such as the Seeed XIAO ESP32S3 Sense or Arduino Nicla Vision, and more powerful single-board computers, like the Raspberry Pi. At its heart lies the Himax WiseEye2 HX6538 processor, featuring a **dual-core Arm Cortex-M55 and an integrated ARM Ethos-U55 neural network unit**.

The Arm Ethos-U55 represents a specialized machine learning processor class, specifically designed as a microNPU to accelerate ML inference in area-constrained embedded and IoT devices. This powerful combination of the Ethos-U55 with the AI-capable Cortex-M55 processor delivers a remarkable 480x uplift in ML performance over existing Cortex-M-based systems. Operating at 400 MHz with configurable internal system memory (SRAM) up to 2.4 MB, the Grove Vision AI V2 offers professional-grade computer vision capabilities while maintaining the power efficiency and compact form factor essential for edge applications.

This positioning makes it an ideal platform for learning advanced TinyML concepts, offering the simplicity and reduced power requirements of smaller systems while providing capabilities that far exceed those of traditional microcontroller-based solutions.

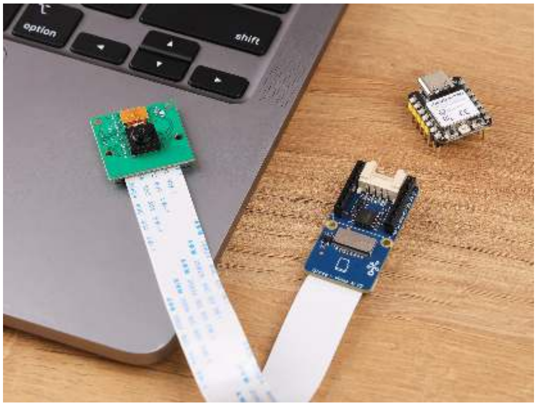


Figure 20.14: Grove - Vision AI Module V2. Source: SEEED Studio

Pre-requisites

- **Grove Vision AI V2 Board**: Ensure you have the Grove Vision AI V2 Board.
- **Raspberry Pi OV5647 Camera Module**: The camera should be connected to the Grove Vision AI V2 Board for image capture.
- **Master Controller**: Can be a Seeed XIAO ESP32S3, a XIAO ESP32C6, or other devices.
- **USB-C Cable**: This is for connecting the board to your computer.
- **Network**: With internet access for downloading the necessary software.
- **XIAO Expansion Board Base**: This helps connect the Master Device to the Physical World (optional).

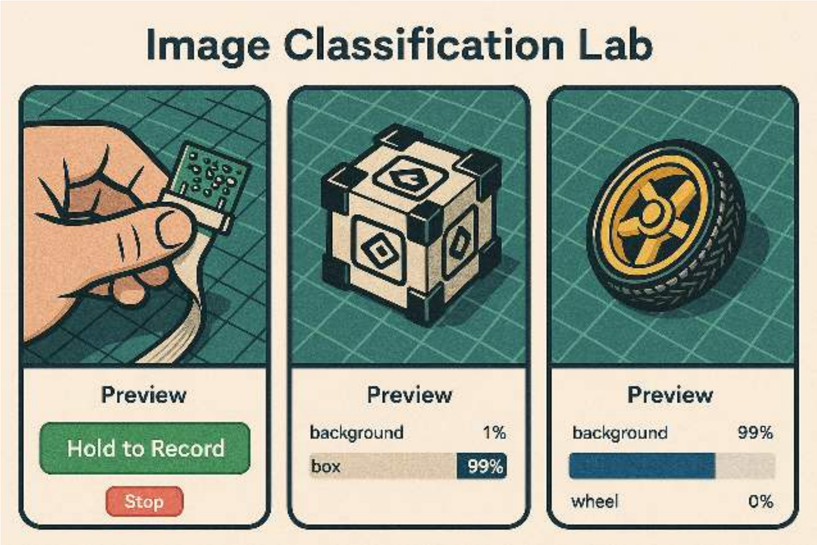
Setup and No-Code Applications

- Setup and No-Code Apps

Exercises

Modality	Task	Description	Link
Vision	Image Classification	Learn to classify images	Link
Vision	Object Detection	Implement object detection	Link

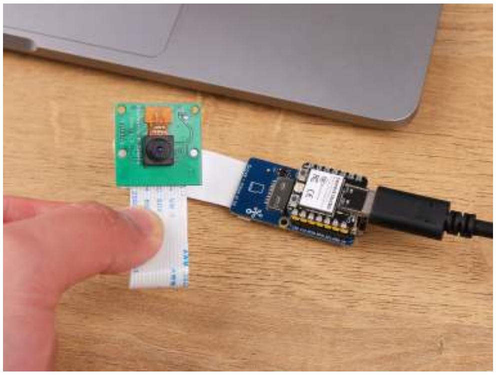
Setup and No-Code Applications



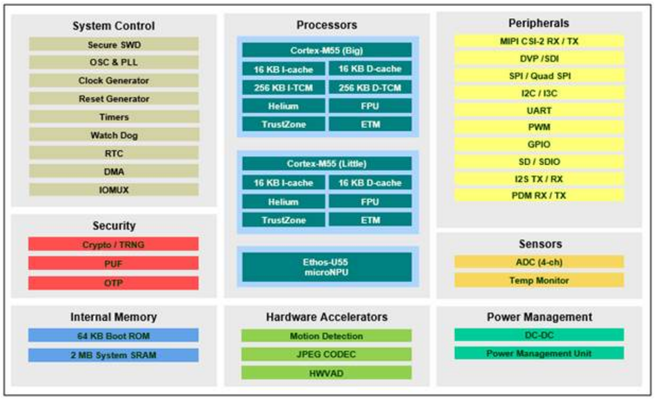
In this Lab, we will explore computer vision (CV) applications using the Seeed Studio *Grove Vision AI Module V2*, a powerful yet compact device specifically designed for embedded machine learning applications. Based on the **Himax WiseEye2** chip, this module is designed to enable AI capabilities on edge devices, making it an ideal tool for Edge Machine Learning (ML) applications.

Introduction

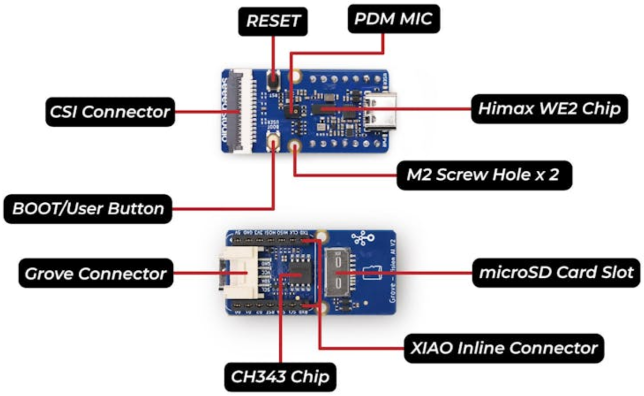
Grove Vision AI Module (V2) Overview



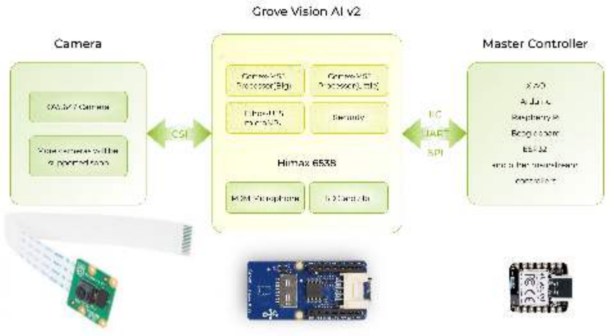
The Grove Vision AI (V2) is an MCU-based vision AI module that utilizes a Himax WiseEye2 HX6538 processor featuring a **dual-core Arm Cortex-M55 and an integrated ARM Ethos-U55 neural network unit**. The Arm Ethos-U55 is a machine learning (ML) processor class, specifically designed as a microNPU, to accelerate ML inference in area-constrained embedded and IoT devices. The Ethos-U55, combined with the AI-capable Cortex-M55 processor, provides a 480x uplift in ML performance over existing Cortex-M-based systems. Its clock frequency is 400 MHz, and its internal system memory (SRAM) is configurable, with a maximum capacity of 2.4 MB.



Note: Based on Seeed Studio documentation, besides the Himax internal memory of 2.5MB (2.4MB SRAM + 64KB ROM), the Grove Vision AI (V2) is also equipped with a 16MB/133 MHz external flash.

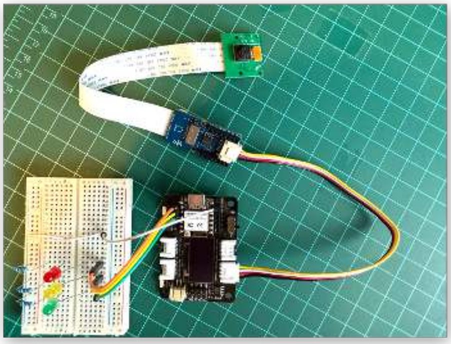


Below is a block Diagram of the Grove Vision AI (V2) system, including a camera and a master controller.

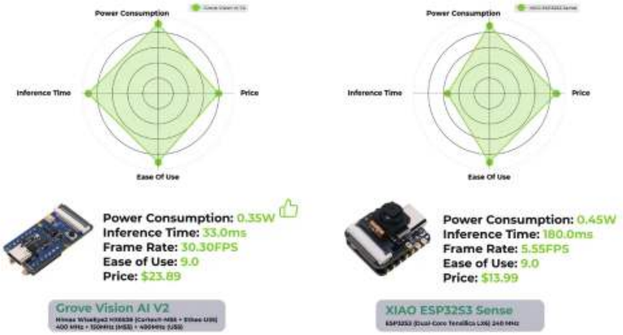


With interfaces like **IIC, UART, SPI, and Type-C,** the Grove Vision AI (V2) can be easily connected to devices such as **XIAO, Raspberry Pi, BeagleBoard, and ESP-based products** for further development. For instance, integrating Grove Vision AI V2 with one of the devices from the XIAO family makes it easy to access the data resulting from inference on the device through the Arduino IDE or MicroPython, and conveniently connect to the cloud or dedicated servers, such as Home Assistance.

> Using the **I2C Grove connector**, the Grove Vision AI V2 can be easily connected with any Master Device.



Besides performance, another area to comment on is **Power Consumption**. For example, in a comparative test against the XIAO ESP32S3 Sense, running Swift-YOLO Tiny 96x96, despite achieving higher performance (30 FPS vs. 5.5 FPS), the Grove Vision AI V2 exhibited lower power consumption (0.35 W vs. 0.45 W) when compared with the XIAO ESP32S3 Sense.

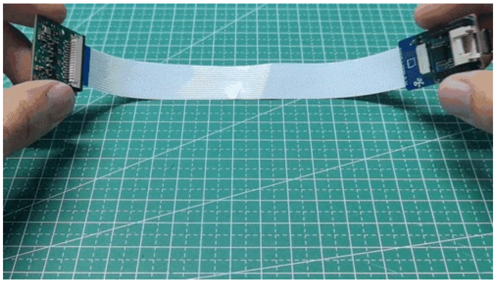


> The above comparison (and with other devices) can be found in the article 2024 MCU AI Vision Boards: Performance Comparison, which confirms the power of Grove Vision AI (V2).

Camera Installation

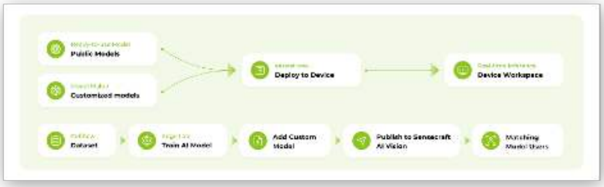
Having the Grove Vision AI (V2) and camera ready, you can connect, for example, a **Raspberry Pi OV5647 Camera Module** via the CSI cable.

> When connecting, please pay attention to the direction of the row of pins and ensure they are plugged in correctly, not in the opposite direction.



The SenseCraft AI Studio

The SenseCraft AI Studio is a robust platform that offers a wide range of AI models compatible with various devices, including the XIAO ESP32S3 Sense and the **Grove Vision AI V2**. In this lab, we will walk through the process of using an AI model with the Grove Vision AI V2 and preview the model's output. We will also explore some key concepts, settings, and how to optimize the model's performance.



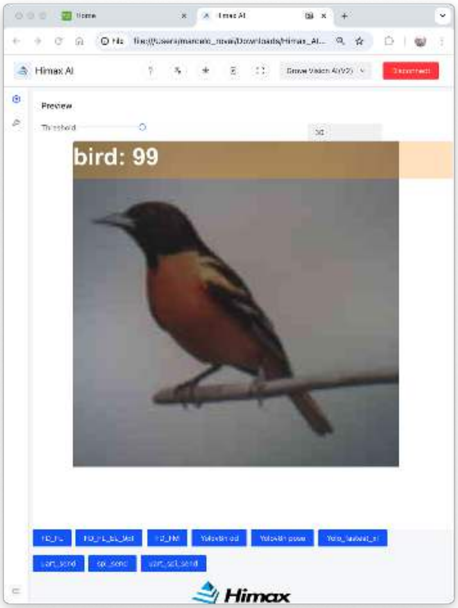
Models can also be deployed using the **SenseCraft Web Toolkit**, a simplified version of the SenseCraft AI Studio.

> We can start using the SenseCraft Web Toolkit for simplicity, or go directly to the SenseCraft AI Studio, which has more resources.

The SenseCraft Web-Toolkit

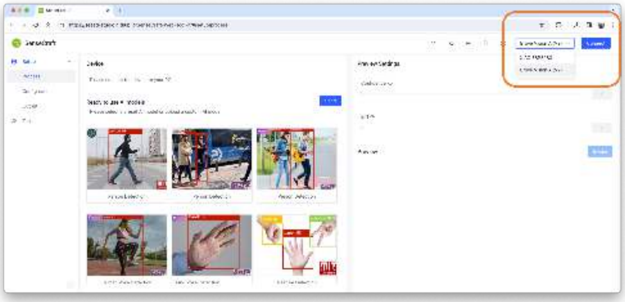
The SenseCraft Web Toolkit is a visual model deployment tool included in the SSCMA(Seeed SenseCraft Model Assistant). This tool enables us to deploy models to various platforms with ease through simple operations. The tool offers a user-friendly interface and does not require any coding.

The SenseCraft Web Toolkit is based on the Himax AI Web Toolkit, which can (**optionally**) be downloaded from here. Once downloaded and unzipped to the local PC, double-click `index.html` to run it locally.

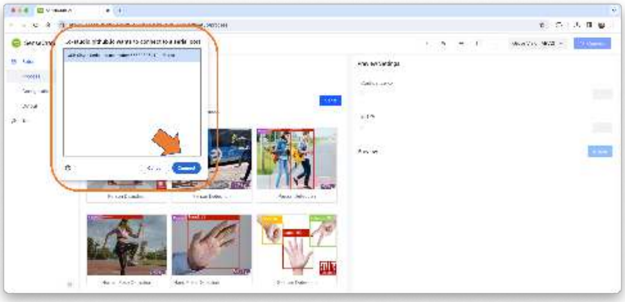


But in our case, let's follow the steps below to start the **SenseCraft-Web-Toolkit**:

- Open the SenseCraft-Web-Toolkit website on a web browser as **Chrome**.
- Connect Grove Vision AI (V2) to your computer using a Type-C cable.
- Having the XIAO connected, select it as below:

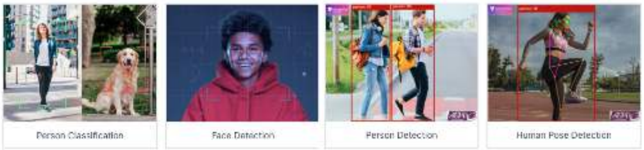


- Select the device/Port and press `[Connect]`:



> Note: The **WebUSB tool** may not function correctly in certain browsers, such as Safari. Use Chrome instead.

We can try several Basic Computer Vision models previously uploaded by Seeed Studio. Passing the cursor over the AI models, we can have some information about them, such as name, description, **category** (Image Classification, Object Detection, or Pose/Keypoint Detection), the **algorithm** (like YOLO V5 or V8, FOMO, MobileNet V2, etc.) and **metrics** (Accuracy or mAP).



We can choose one of those ready-to-use AI models by clicking on it and pressing the `[Send]` button, or upload our model.

For the **SenseCraft AI** platform, follow the instructions here.

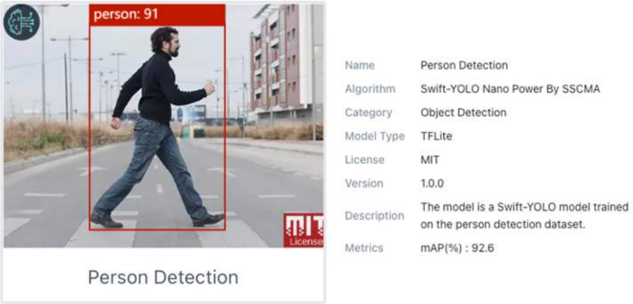
Exploring CV AI models

Object Detection

Object detection is a pivotal technology in computer vision that focuses on identifying and locating objects within digital images or video frames. Unlike image classification, which categorizes an entire image into a single label, object detection recognizes multiple objects within the image and determines their precise locations, typically represented by bounding boxes. This capability is crucial for a wide range of applications, including autonomous vehicles, security, surveillance systems, and augmented reality, where understanding the context and content of the visual environment is essential.

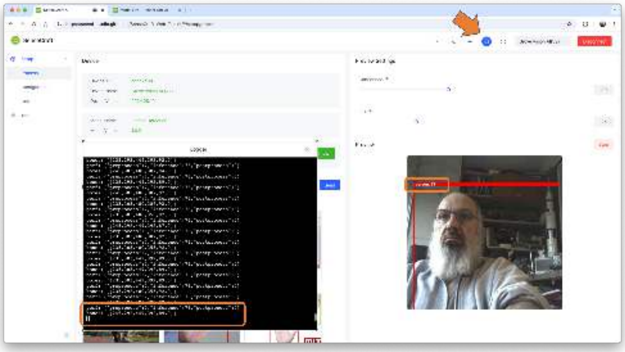
Common architectures that have set the benchmark in object detection include the YOLO (You Only Look Once), SSD (Single Shot MultiBox Detector), FOMO (Faster Objects, More Objects), and Faster R-CNN (Region-based Convolutional Neural Networks) models.

Let's choose one of the ready-to-use AI models, such as **Person Detection**, which was trained using the Swift-YOLO algorithm.



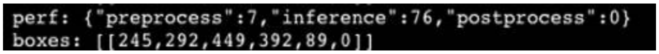
Name	Person Detection
Algorithm	Swift-YOLO Nano Power By SSCMA
Category	Object Detection
Model Type	TFLite
License	MIT
Version	1.0.0
Description	The model is a Swift-YOLO model trained on the person detection dataset.
Metrics	mAP(%) : 92.6

Once the model is uploaded successfully, you can see the live feed from the Grove Vision AI (V2) camera in the Preview area on the right. Also, the inference details can be shown on the Serial Monitor by clicking on the `[Device Log]` button at the top.



> In the SenseCraft AI Studio, the Device Logger is always on the screen.

Pointing the camera at me, only one person was detected, so that the model output will be a single "box". Looking in detail, the module sends continuously two lines of information:



```
perf: {"preprocess":7,"inference":76,"postprocess":0}
boxes: [[245,292,449,392,89,0]]
```

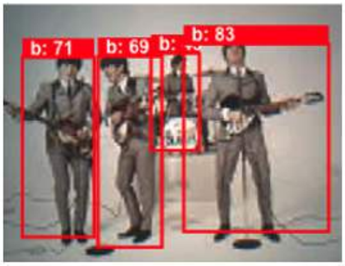
perf (Performance), displays latency in milliseconds.

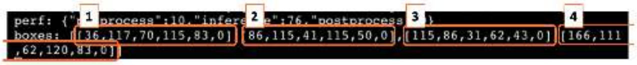
- Preprocess time (image capture and Crop): **7ms**;
- Inference time (model latency): **76ms (13 fps)**
- Postprocess time (display of the image and inclusion of data): less than 0ms.

boxes: Show the objects detected in the image. In this case, only one.

- The box has the x, y, w, and h coordinates of (**245**, **292**,**449**,**392**), and the object (person, label **0**) was captured with a value of **.89**.

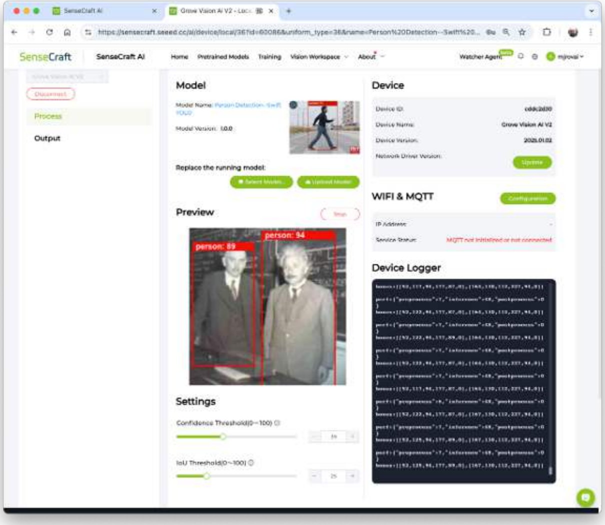
If we point the camera at an image with several people, we will get one box for each person (object):





```
perf: {" process":10,"infere e":76,"postprocess }
boxes: [[36,117,70,115,83,0] 86,115,41,115,50,0],[115,86,31,62,43,0] [166,111
,62,120,83,0]
```

On the SenseCraft AI Studio, the inference latency (48ms) is lower than on the SenseCraft ToolKit (76ms), due to a distinct deployment implementation.



Model Name: Person Detection--Swift YOLO
Model Version: 1.0.0
Device Name: Grove Vision AI V2
Device Version: 2025.01.02
Network Driver Version:
IP Address: -
Service Status: MQTT not initialized or not connected
Confidence Threshold(0~100)
IoU Threshold(0~100)

Power Consumption

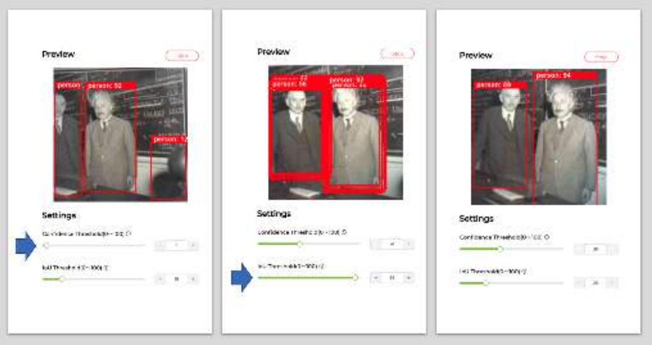
The peak power consumption running this Swift-YOLO model was 410 milliwatts.

Preview Settings

We can see that in the Settings, two settings options can be adjusted to optimize the model's recognition accuracy.

- **Confidence:** Refers to the level of certainty or probability assigned to its predictions by a model. This value determines the minimum confidence level required for the model to consider a detection as valid. A higher confidence threshold will result in fewer detections but with higher certainty, while a lower threshold will allow more detections but may include some false positives.

- **IoU:** Used to assess the accuracy of predicted bounding boxes compared to truth bounding boxes. IoU is a metric that measures the overlap between the predicted bounding box and the ground truth bounding box. It is used to determine the accuracy of the object detection. The IoU threshold sets the minimum IoU value required for a detection to be considered a true positive. Adjusting this threshold can help in fine-tuning the model's precision and recall.



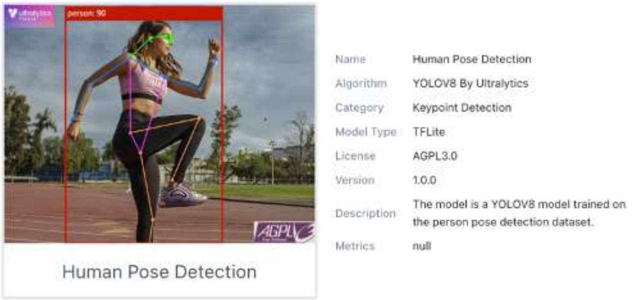
Experiment with different values for the Confidence Threshold and IoU Threshold to find the optimal balance between detecting persons accurately and minimizing false positives. The best settings may vary depending on our specific application and the characteristics of the images or video feed.

Pose/Keypoint Detection

Pose or keypoint detection is a sophisticated area within computer vision that focuses on identifying specific points of interest within an image or video frame, often related to human bodies, faces, or other objects of interest. This technology can detect and map out the various keypoints of a subject, such as the **joints on a human body** or the features of a face, enabling the analysis of postures, movements, and gestures. This has profound implications for various applications, including augmented reality, human-computer interaction, sports analytics, and healthcare monitoring, where understanding human motion and activity is crucial.

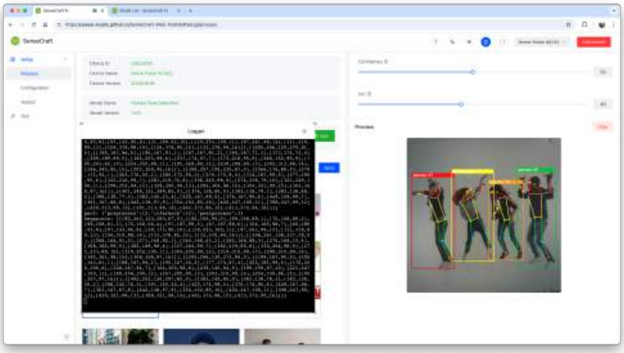
Unlike general object detection, which identifies and locates objects, pose detection drills down to a finer level of detail, capturing the nuanced positions and orientations of specific parts. Leading architectures in this field include OpenPose, AlphaPose, and PoseNet, each designed to tackle the challenges of pose estimation with varying degrees of complexity and precision. Through advancements in deep learning and neural networks, pose detection has become increasingly accurate and efficient, offering real-time insights into the intricate dynamics of subjects captured in visual data.

So, let's explore this popular CV application, *Pose/Keypoint Detection*.



Name	Human Pose Detection
Algorithm	YOLOV8 By Ultralytics
Category	Keypoint Detection
Model Type	TFLite
License	AGPL3.0
Version	1.0.0
Description	The model is a YOLOV8 model trained on the person pose detection dataset.
Metrics	null

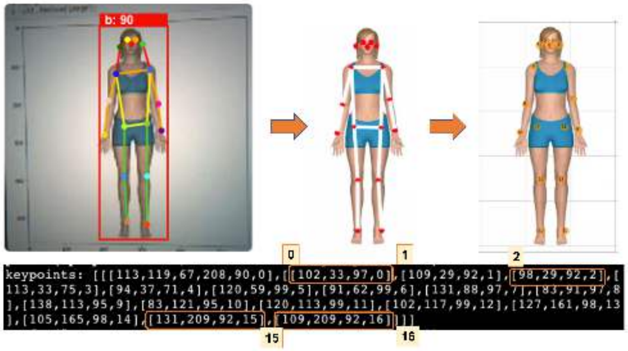
Stop the current model inference by pressing `[Stop]` in the Preview area. Select the model and press `[Send]`. Once the model is uploaded successfully, you can view the live feed from the Grove Vision AI (V2) camera in the Preview area on the right, along with the inference details displayed in the Serial Monitor (accessible by clicking the `[Device Log]` button at the top).



The YOLOV8 Pose model was trained using the COCO-Pose Dataset, which contains 200K images labeled with **17** keypoints for pose estimation tasks.

Let's look at a single screenshot of the inference (to simplify, let's analyse an image with a single person in it). We can note that we have two lines, one with the inference **performance** in milliseconds (121 ms) and a second line with the **keypoints** as below:

- 1 box of info, the same as we got with the object detection example (box coordinates (113, 119, 67, 208), inference result (90), label (0).
- 17 groups of 4 numbers represent the 17 "joints" of the body, where '0' is the nose, '1' and '2' are the eyes, '15' and' 16' are the feet, and so on.



```
keypoints: [[[113,119,67,208,90,0],[[102,33,97,0],[109,29,92,1],[98,29,92,2],[
113,33,75,3],[94,37,71,4],[120,59,99,5],[91,62,99,6],[131,88,97,7],[83,91,97,8
],[138,113,95,9],[83,121,95,10],[120,113,99,11],[102,117,99,12],[127,161,98,13
],[105,165,98,14],[131,209,92,15],[109,209,92,16]]]]
```

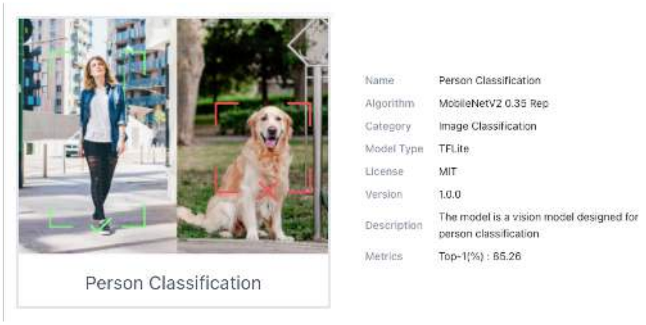
To understand a pose estimation project more deeply, please refer to the tutorial: Exploring AI at the Edge! - Pose Estimation.

Image Classification

Image classification is a foundational task within computer vision aimed at categorizing **entire images** into one of several predefined classes. This process involves analyzing the visual content of an image and assigning it a label from a fixed set of categories based on the predominant object or scene it contains.

Image classification is crucial in various applications, ranging from organizing and searching through large databases of images in digital libraries and social media platforms to enabling autonomous systems to comprehend their surroundings. Common architectures that have significantly advanced the field of image classification include Convolutional Neural Networks (CNNs), such as AlexNet, VGGNet, and ResNet. These models have demonstrated remarkable accuracy on challenging datasets, such as **ImageNet,** by learning hierarchical representations of visual data.

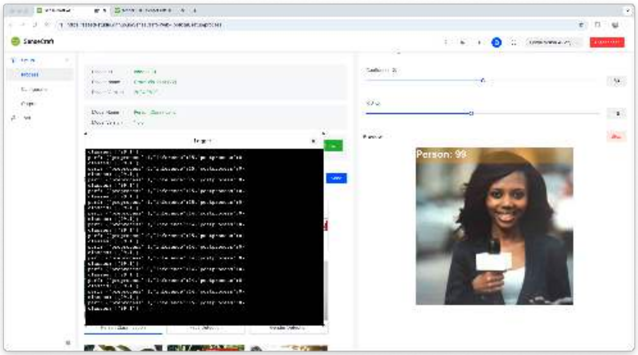
As the cornerstone of many computer vision systems, image classification drives innovation, laying the groundwork for more complex tasks like object detection and image segmentation, and facilitating a deeper understanding of visual data across various industries. So, let's also explore this computer vision application.



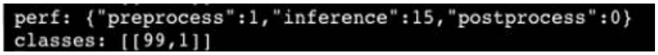
Name	Person Classification
Algorithm	MobileNetV2 0.35 Rep
Category	Image Classification
Model Type	TFLite
License	MIT
Version	1.0.0
Description	The model is a vision model designed for person classification
Metrics	Top-1(%) : 85.26

> This example is available on the SenseCraft ToolKit, but not in the SenseCraft AI Studio. In the last one, it is possible to find other examples of Image Classification.

After the model is uploaded successfully, we can view the live feed from the Grove Vision AI (V2) camera in the Preview area on the right, along with the inference details displayed in the Serial Monitor (by clicking the `[Device Log]` button at the top).



As a result, we will receive a score and the class as output.



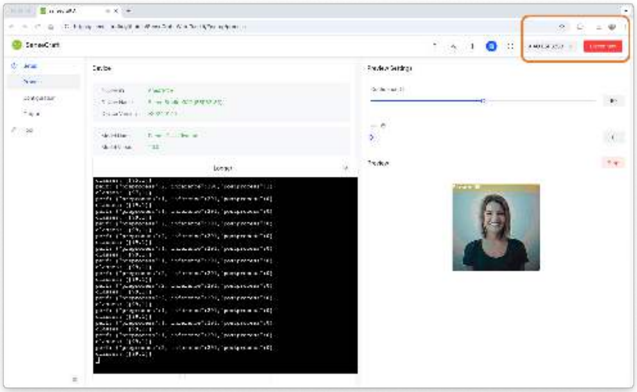
```
perf: {"preprocess":1,"inference":15,"postprocess":0}
classes: [[99,1]]
```

For example, **[99, 1]** means class: 1 (Person) with a score of 0.99. Once this model is a binary classification, class 0 will be "No Person" (or Background). The Inference latency is **15ms** or around 70fps.

Power Consumption

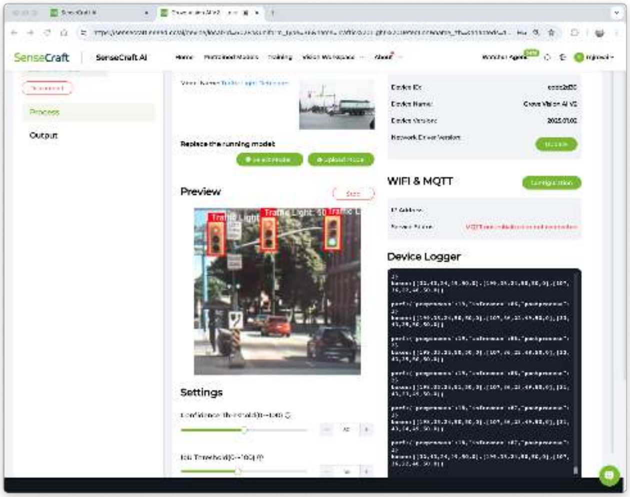
To run the Mobilenet V2 0.35, the Grove Vision AI V2 had a peak current of 80mA at 5.24V, resulting in a **power consumption of 420mW**.

Running the same model on XIAO ESP32S3 Sense, the **power consumption was 523mW** with a latency of 291ms.



Exploring Other Models on SenseCraft AI Studio

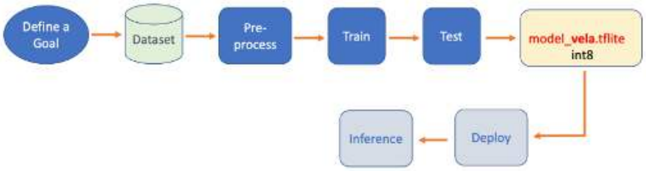
Several public AI models can also be downloaded from the SenseCraft AI WebPage. For example, you can run a Swift-YOLO model, detecting traffic lights as shown here:



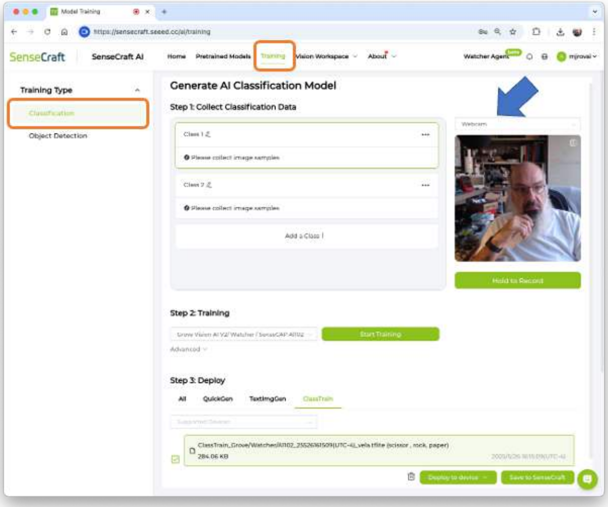
The latency of this model is approximately 86 ms, with an average power consumption of 420 mW.

An Image Classification Project

Let's create a complete Image Classification project, using the SenseCraft AI Studio.



On SenseCraft AI Studio: Let's open the tab Training:



Generate AI Classification Model

Step 1: Collect Classification Data

Class 1

Please collect image samples

Class 2

Please collect image samples

Add a Class

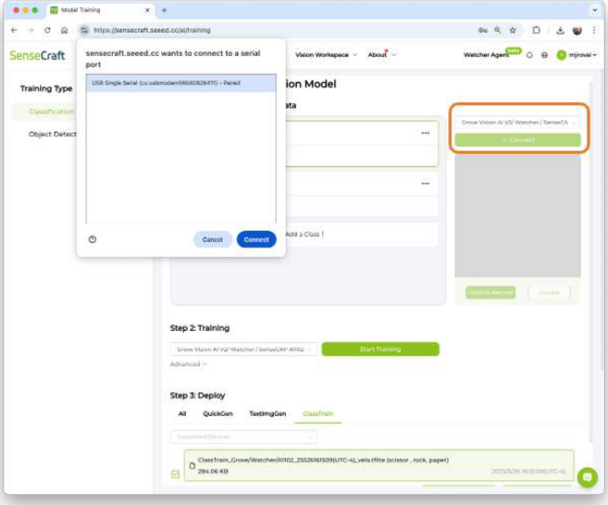
Step 2: Training

Grove Vision AI V2/ Watcher / SenseCAP AI102

Step 3: Deploy

ClassTrain_Grove/Watcher/AI102_25526161509(UTC-4)_vela tflite (scissor , rock, paper)
284.06 KB

The default is to train a `Classification` model with a WebCam if it is available. Let's select the Grove Vision AI V2 instead. Pressing the green button `[Connect]`, a Pop-Up window will appear. Select the corresponding Port and press the blue button `[Connect]`.



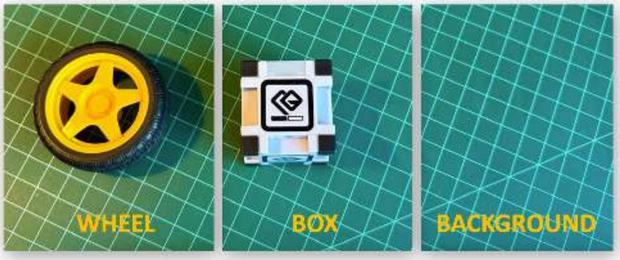
sensecraft.seeed.cc wants to connect to a serial port

USB Single Serial (cu.usbmodem58580826411) - Paired

The image streamed from the Grove Vision AI V2 will be displayed.

The Goal

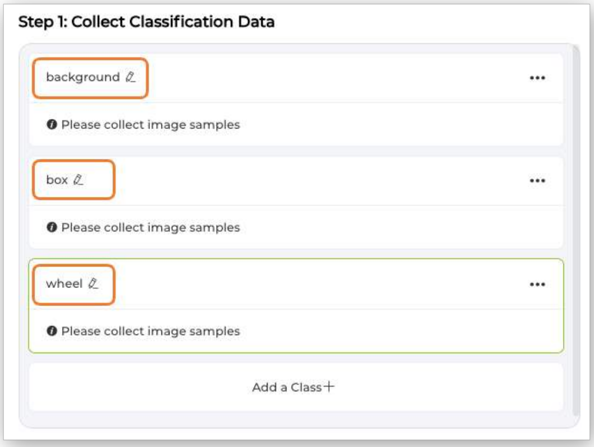
The first step is always to define a goal. Let's classify, for example, two simple objects—for instance, a toy `box` and a toy `wheel`. We should also include a 3rd class of images, `background`, where no object is in the scene.



Data Collection

Let's create the classes, following, for example, an alphabetical order:

- Class1: background
- Class 2: box
- Class 3: wheel



Step 1: Collect Classification Data

background

Please collect image samples

box

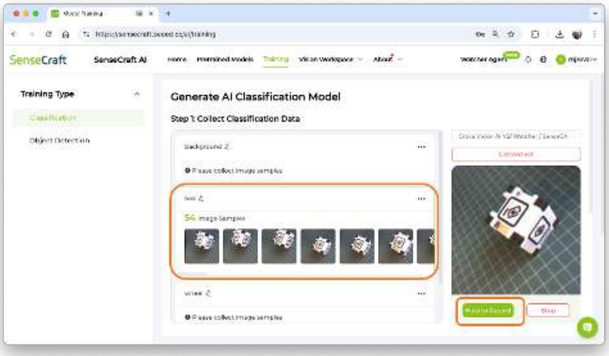
Please collect image samples

wheel

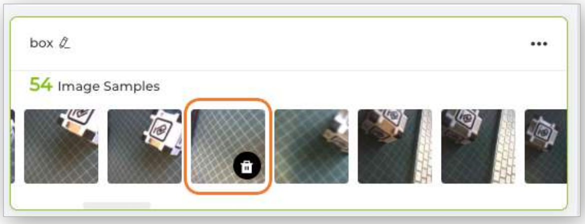
Please collect image samples

Add a Class +

Select one of the classes and keep pressing the green button under the preview area. The collected images will appear on the Image Samples Screen.



After collecting the images, review them and delete any incorrect ones.



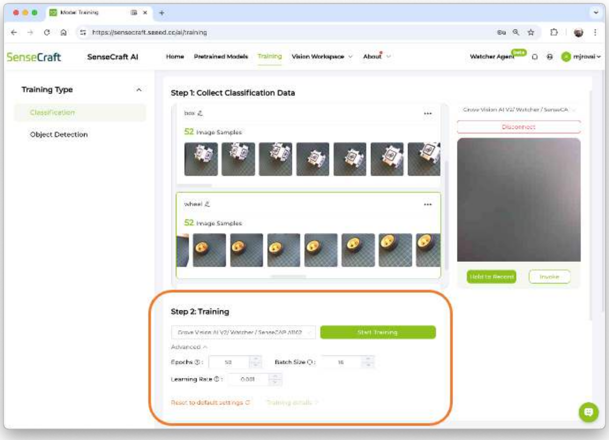
box

54 Image Samples

Collect around 50 images from each class and go to Training Step:

Training

Confirm if the correct device is selected (`Grove Vision AI V2`) and press `[Start Training]`



Step 1: Collect Classification Data

box: 52 Image Samples

wheel: 52 Image Samples

Step 2: Training

Grove Vision AI V2/ Watcher / SenseCAP A1102

Epochs: 50

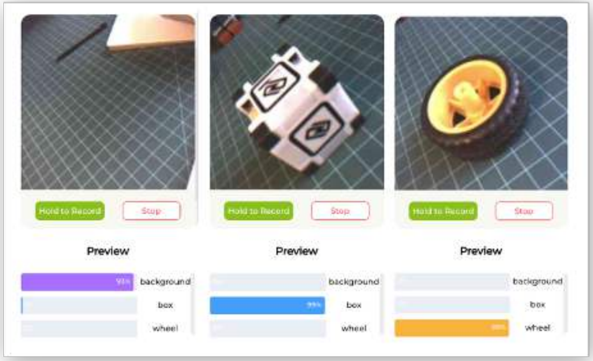
Batch Size: 16

Learning Rate: 0.001

Test

After training, the inference result can be previewed.

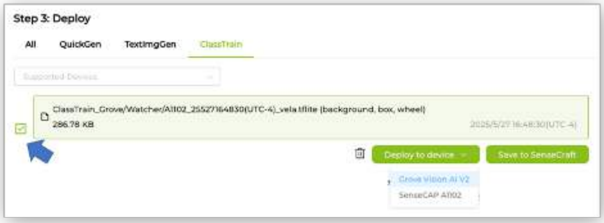
> Note that the model is not running on the device. We are, in fact, only capturing the images with the device and performing a live preview using the training model, which is running in the Studio.



Now is time to really deploy the model in the device:

Deployment

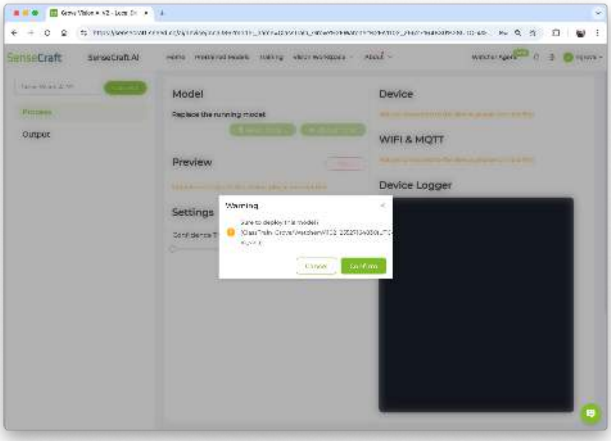
Select the trained model on `[Deploy to device]`, select the Grove Vision AI V2:



Step 3: Deploy

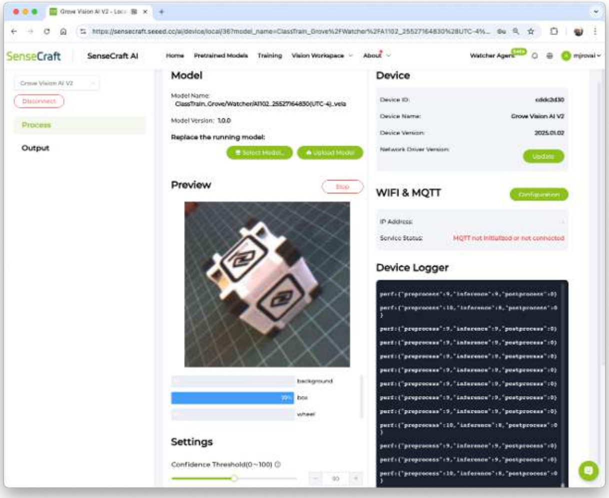
ClassTrain_Grove/Watcher/AI102_25527164830(UTC-4)_vela.tflite (background, box, wheel)
286.78 KB
2025/5/27 16:48:30(UTC-4)

The Studio will redirect us to the `Vision Workplace` tab. Confirm the deployment, select the appropriate Port, and connect it:



The model will be flashed into the device. After an automatic reset, the model will start running on the device. On the Device Logger, we can see that the inference has a **latency of approximately 8 ms**, corresponding to a **frame rate of 125 frames per second (FPS)**.

Also, note that it is possible to adjust the model's confidence.



Model

Model Name: ClassTrain_Grove/Watcher/AI102_25527164830(UTC-4)_vela

Model Version: 1.0.0

Replace the running model:

Device

Device ID: cdda2d30

Device Name: Grove Vision AI V2

Device Version: 2025.01.02

Network Driver Version:

Preview

WIFI & MQTT

IP Address:

Service Status: MQTT not initialized or not connected

Device Logger

```
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":10,"inference":8,"postprocess":0
}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":10,"inference":8,"postprocess":0
}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":9,"inference":9,"postprocess":0}
perf:{"preprocess":10,"inference":8,"postprocess":0
}
```

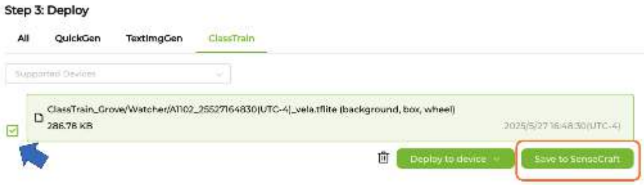
Settings

Confidence Threshold(0~100): 50

To run the Image Classification Model, the Grove Vision AI V2 had a peak current of 80mA at 5.24V, resulting in a **power consumption of 420mW**.

Saving the Model

It is possible to save the model in the SenseCraft AI Studio. The Studio will keep all our models, which can be deployed later. For that, return to the `Training` tab and select the button `[Save to SenseCraft]`:



Step 3: Deploy

☑ ClassTrain_Grove/Watcher/AI102_25527164830(UTC-4)_vela.tflite (background, box, wheel)
286.78 KB 2025/5/27 16:48:30(UTC-4)

Conclusion

In this lab, we explored several computer vision (CV) applications using the Seeed Studio Grove Vision AI Module V2, demonstrating its exceptional capabilities as a powerful yet compact device specifically designed for embedded machine learning applications.

Performance Excellence: The Grove Vision AI V2 demonstrated remarkable performance across multiple computer vision tasks. With its **Himax WiseEye2 chip** featuring a **dual-core Arm Cortex-M55 and integrated ARM Ethos-U55 neural network unit**, the device delivered:

- **Image Classification**: **15 ms** inference time (67 FPS)

- **Object Detection (Person)**: **48 ms to 76 ms** inference time (21 FPS to 13 FPS)
- **Pose Detection**: **121 ms** real-time keypoint detection with 17-joint tracking (8 FPS)

Power Efficiency Leadership: One of the most compelling advantages of the Grove Vision AI V2 is its superior power efficiency. Comparative testing revealed significant improvements over traditional embedded platforms:

- **Grove Vision AI V2**: 80 mA (**410 mW**) peak consumption (60+ FPS)
- **XIAO ESP32S3**: Performing similar CV tasks (Image Classification) **523 mW** (3+ FPS)

Practical Implementation: The device's versatility was demonstrated through a comprehensive end-to-end project, encompassing dataset creation, model training, deployment, and offline inference.

Developer-Friendly Ecosystem: The SenseCraft AI Studio, with its no-code deployment and integration capabilities for custom applications, makes the Grove Vision AI V2 accessible to both beginners and advanced developers. The extensive library of pre-trained models and support for custom model deployment provide flexibility for diverse applications.

The Grove Vision AI V2 represents a significant advancement in edge AI hardware, offering professional-grade computer vision capabilities in a compact, energy-efficient package that democratizes AI deployment for embedded applications across industrial, IoT, and educational domains.

Key Takeaways

This Lab demonstrates that sophisticated computer vision applications are not limited to cloud-based solutions or power-hungry hardware, as the Raspberry Pi or Jetson Nanos – they can now be deployed effectively at the edge with remarkable efficiency and performance.

Optionally, we can have the XIAO Vision AI Camera. This innovative vision solution seamlessly combines the Grove Vision AI V2 module, XIAO ESP32-C3 controller, and an OV5647 camera, all housed in a custom 3D-printed enclosure:

Resources

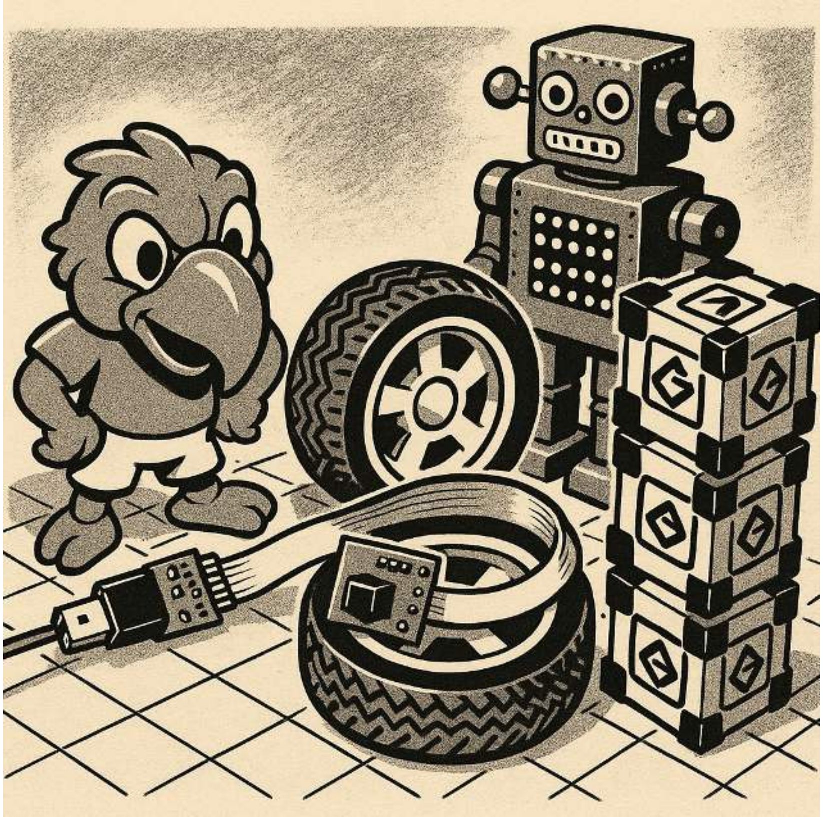
SenseCraft AI Studio Instructions.

SenseCraft-Web-Toolkit website.

SenseCraft AI Studio
Himax AI Web Toolkit
Himax examples

Image Classification

Using Seeed Studio Grove Vision AI Module V2 (Himax WiseEye2)

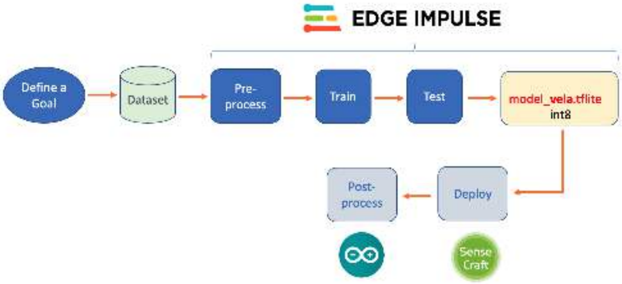


In this Lab, we will explore Image Classification using the Seeed Studio *Grove Vision AI Module V2*, a powerful yet compact device specifically designed for embedded machine learning applications. Based on the **Himax WiseEye2** chip, this module is designed to enable AI capabilities on edge devices, making it an ideal tool for Edge Machine Learning (ML) applications.

Introduction

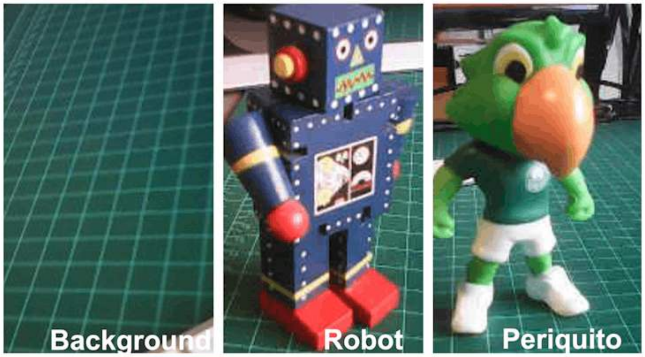
So far, we have explored several computer vision models previously uploaded by Seeed Studio or used the SenseCraft AI Studio for Image Classification, without choosing a specific model. Let's now develop our Image Classification project from scratch, where we will select our data and model.

Below, we can see the project's main steps and where we will work with them:



Project Goal

The first step in any machine learning (ML) project is defining the goal. In this case, the goal is to detect and classify two specific objects present in a single image. For this project, we will use two small toys: a robot and a small Brazilian parrot (named *Periquito*). Also, we will collect images of a background where those two objects are absent.

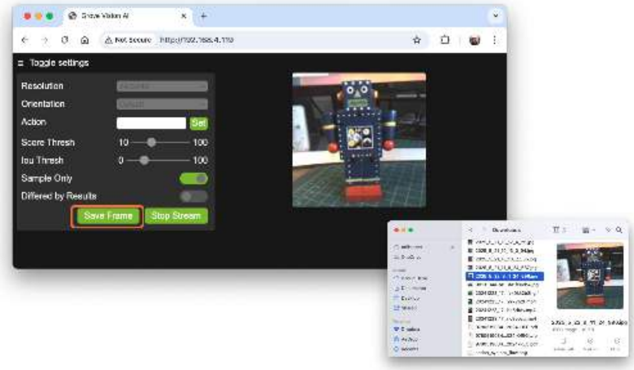


Data Collection

With the Machine Learning project goal defined, dataset collection is the next and most crucial step. Suppose your project utilizes images that are publicly available on datasets, for example, to be used on a **Person Detection** project. In that case, you can download the Wake Vision dataset for use in the project.

But, in our case, we define a project where the images do not exist publicly, so we need to generate them. We can use a phone, computer camera, or other devices to capture the photos, offline or connected to the Edge Impulse Studio.

If you want to use the Grove Vision AI V2 to capture your dataset, you can use the SenseCraft AI Studio as we did in the previous Lab, or the `camera_web_server` sketch as we will describe later in the **Postprocessing / Getting the Video Stream** section of this Lab.

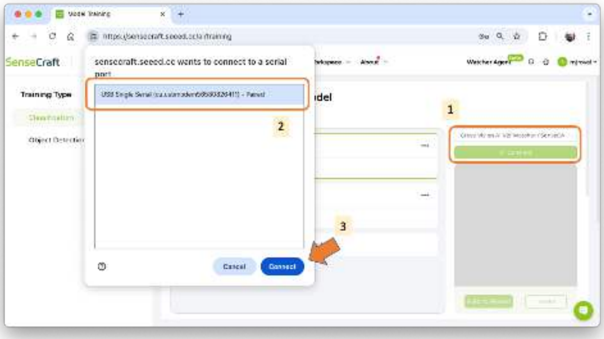


In this Lab, we will use the SenseCraft AI Studio to collect the dataset.

Collecting Data with the SenseCraft AI Studio

On SenseCraft AI Studio: Let's open the tab Training.

The default is to train a `Classification` model with a WebCam if it is available. Let's select the `Grove Vision AI V2 instead`. Pressing the green button `[Connect]` **(1),** a Pop-Up window will appear. Select the corresponding Port **(2)** and press the blue button `[Connect]` **(3)**.

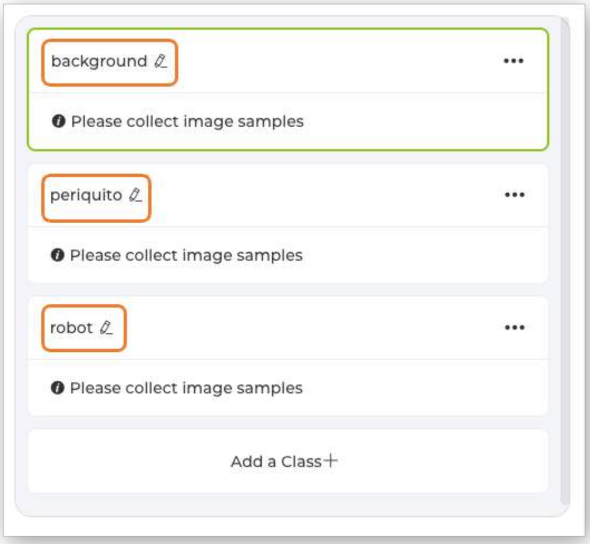


The image streamed from the Grove Vision AI V2 will be displayed.

Image Collection

Let's create the classes, following, for example, an alphabetical order:

- Class1: background
- Class 2: periquito
- Class 3: robot



background ✎ •••

Please collect image samples

periquito ✎ •••

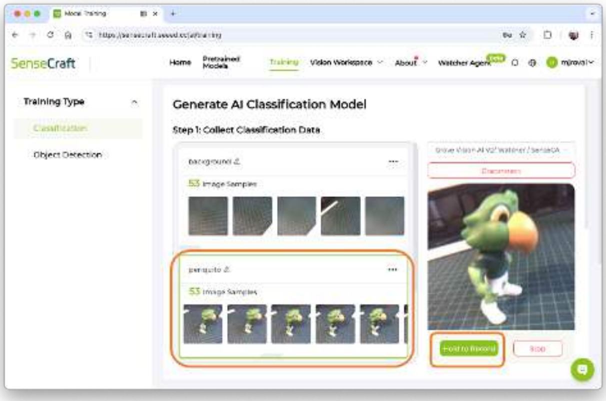
Please collect image samples

robot ✎ •••

Please collect image samples

Add a Class +

Select one of the classes (note that a green line will be around the window) and keep pressing the green button under the preview area. The collected images will appear on the Image Samples Screen.



Generate AI Classification Model

Step 1: Collect Classification Data

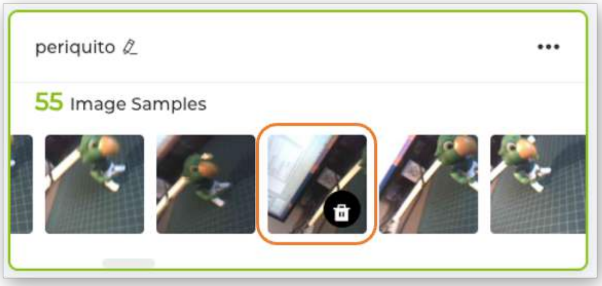
background ✎ •••

53 Image Samples

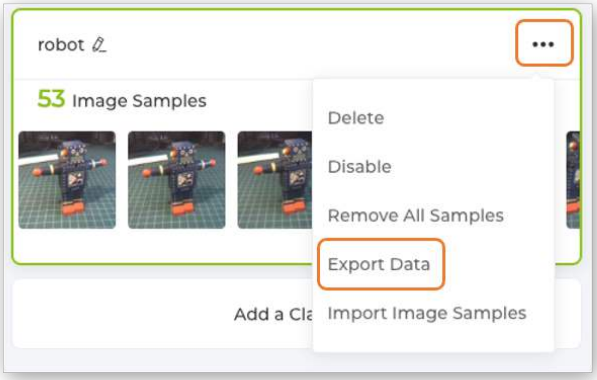
periquito ✎ •••

53 Image Samples

After collecting the images, review them and, if necessary, delete any incorrect ones.



Collect around 50 images from each class. After you collect the three classes, open the menu on each of them and select `Export Data`.

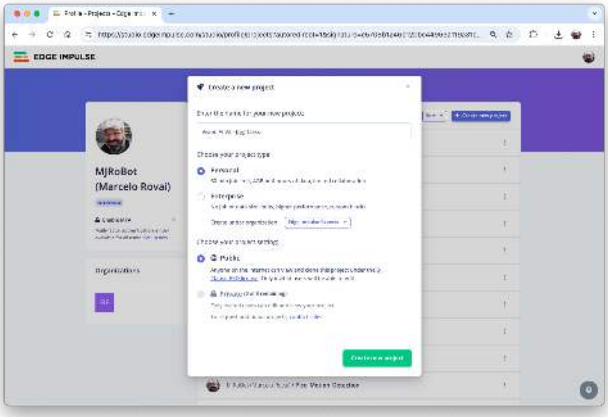


In the Download area of the Computer, we will get three zip files, each one with its corresponding class name. Each Zip file contains a folder with the images.

Uploading the dataset to the Edge Impulse Studio

We will use the Edge Impulse Studio to train our model. Edge Impulseis a leading development platform for machine learning on edge devices.

- Enter your account credentials (or create a free account) at Edge Impulse.
- Next, create a new project:



Create a new project

Enter the name for your new project:

Choose your project type

● Personal

○ Enterprise

Choose your project setting

● Public

○ Private

Create new project

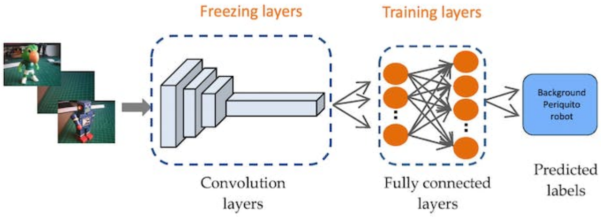
The dataset comprises approximately 50 images per label, with 40 for training and 10 for testing.

Impulse Design and Pre-Processing

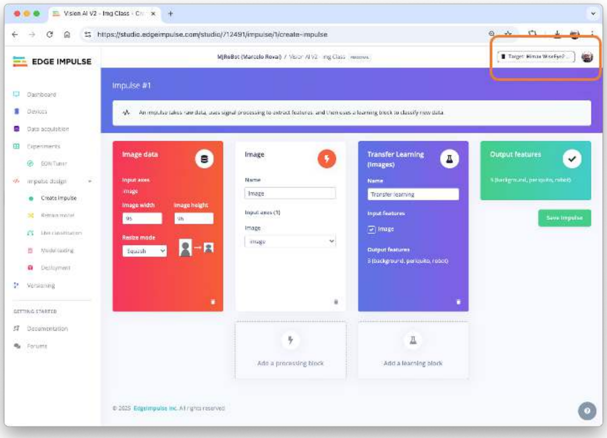
Impulse Design

An impulse takes raw data (in this case, images), extracts features (resizes pictures), and then uses a learning block to classify new data.

Classifying images is the most common application of deep learning, but a substantial amount of data is required to accomplish this task. We have around 50 images for each category. Is this number enough? Not at all! We will need thousands of images to "teach" or "model" each class, allowing us to differentiate them. However, we can resolve this issue by retraining a previously trained model using thousands of images. We refer to this technique as "Transfer Learning" (TL). With TL, we can fine-tune a pre-trained image classification model on our data, achieving good performance even with relatively small image datasets, as in our case.



So, starting from the raw images, we will resize them (96x96) pixels and feed them to our Transfer Learning block:



An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

Input axes: image
Image width: 96
Image height: 96
Resize mode: Squash

Image — Name: Image
Input axes (1): Image — image

Transfer Learning (Images) — Name: Transfer learning
Input features: ☑ Image
Output features: 3 (background, periquito, robot)

Output features: 3 (background, periquito, robot)

For comparison, we will keep the image size as 96 x 96. However, keep in mind that with the Grove Vision AI Module V2 and its internal SRAM of 2.4 MB, larger images can be utilized (for example, 160 x 160).

Also select the `Target` device (`Himax WiseEye2 (M55 400 MHz + U55)`) on the up-right corner.

Pre-processing (Feature generation)

Besides resizing the images, we can convert them to grayscale or retain their original RGB color depth. Let's select `[RGB]` in the `Image` section. Doing that, each data sample will have a dimension of 27,648 features (96x96x3). Pressing `[Save Parameters]` will open a new tab, `Generate Features`. Press the button `[Generate Features]` to generate the features.

Model Design, Training, and Test

In 2007, Google introduced MobileNetV1. In 2018, MobileNetV2: Inverted Residuals and Linear Bottlenecks, was launched, and, in 2019, the V3. The Mobilinet is a family of general-purpose computer vision neural networks explicitly designed for mobile devices to support classification, detection, and other applications. MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of various use cases.

Although the base MobileNet architecture is already compact and has low latency, a specific use case or application may often require the model to be even smaller and faster. MobileNets introduce a straightforward parameter, α (alpha), called the width multiplier to construct these smaller, less computationally expensive models. The role of the width multiplier α is to thin a network uniformly at each layer.

Edge Impulse Studio has available MobileNet V1 (96x96 images) and V2 (96x96 and 160x160 images), with several different α values (from 0.05 to 1.0).

For example, you will get the highest accuracy with V2, 160x160 images, and α=1.0. Of course, there is a trade-off. The higher the accuracy, the more memory (around 1.3M RAM and 2.6M ROM) will be needed to run the model, implying more latency. The smaller footprint will be obtained at another extreme with MobileNet V1 and α=0.10 (around 53.2K RAM and 101K ROM).

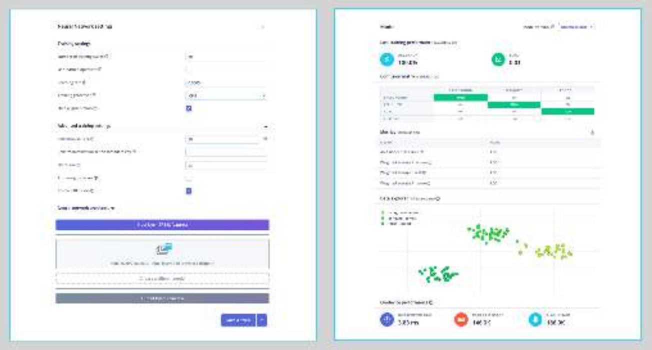
> For comparison, we will use the **MobileNet V2 0.1** as our base model (but a model with a greater alpha can be used here). The final layer of our model, preceding the output layer, will have 8 neurons with a 10% dropout rate for preventing overfitting.

Another necessary technique to use with deep learning is **data augmentation**. Data augmentation is a method that can help improve the accuracy of machine learning models by creating additional artificial data. A data augmentation system makes small, random changes to your training data during the training process (such as flipping, cropping, or rotating the images).

Set the Hyperparameters:

- Epochs: 20,
- Bach Size: 32
- Learning Rate: 0.0005
- Validation size: 20%

Training result:



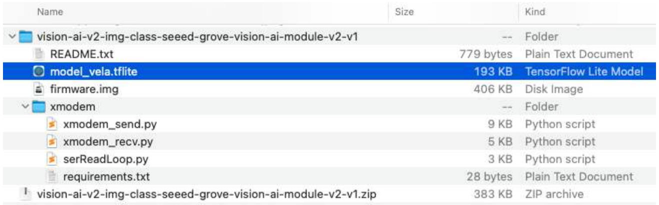
The model profile predicts **146 KB of RAM and 187 KB of Flash**, indicating no problem with the Grove AI Vision (V2), which has almost 2.5 MB of internal SRAM. Additionally, the Studio indicates a **latency of around 4 ms.**

> Despite this, with a 100% accuracy on the Validation set when using the spare data for testing, we confirmed an Accuracy of 81%, using the Quantized (Int8) trained model. However, it is sufficient for our purposes in this lab.

Model Deployment

On the Deployment tab, we should select: `Seeed Grove Vision AI Module V2 (Himax WiseEye2)` and press `[Build]`. A ZIP file will be downloaded to our computer.

The Zip file contains the `model_vela.tflite`, which is a TensorFlow Lite (TFLite) model optimized for neural processing units (NPUs) using the Vela compiler, a tool developed by Arm to adapt TFLite models for Ethos-U NPUs.

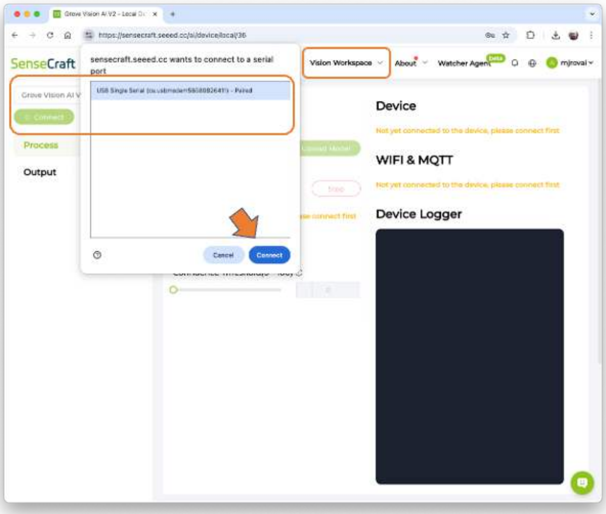


Name	Size	Kind
vision-ai-v2-img-class-seeed-grove-vision-ai-module-v2-v1	--	Folder
README.txt	779 bytes	Plain Text Document
model_vela.tflite	193 KB	TensorFlow Lite Model
firmware.img	406 KB	Disk Image
xmodem	--	Folder
xmodem_send.py	9 KB	Python script
xmodem_recv.py	5 KB	Python script
serReadLoop.py	3 KB	Python script
requirements.txt	28 bytes	Plain Text Document
vision-ai-v2-img-class-seeed-grove-vision-ai-module-v2-v1.zip	383 KB	ZIP archive

We can flash the model following the instructions in the `README.txt` or use the SenseCraft AI Studio. We will use the latter.

Deploy the model on the SenseCraft AI Studio

On SenseCraft AI Studio, go to the `Vision Workspace` tab, and connect the device:

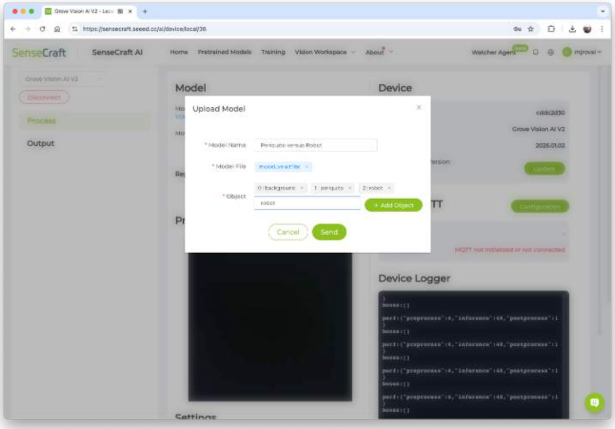


sensecraft.seeed.cc wants to connect to a serial port

USB Single Serial (cu.usbmodem58180826411) - Paired

Not yet connected to the device, please connect first

You should see the last model that was uploaded to the device. Select the green button `[Upload Model]`. A pop-up window will ask for the **model name**, the **model file,** and to enter the class names (**objects**). We should use labels following alphabetical order: `0: background`, `1: periquito`, and `2: robot`, and then press `[Send]`.



Upload Model

* Model Name: Periquito versus Robot

* Model File: model.vela.tflite

* Object: 0: background, 1: periquito, 2: robot

robot

+ Add Object

Cancel Send

cddc2d30

Grove Vision AI V2

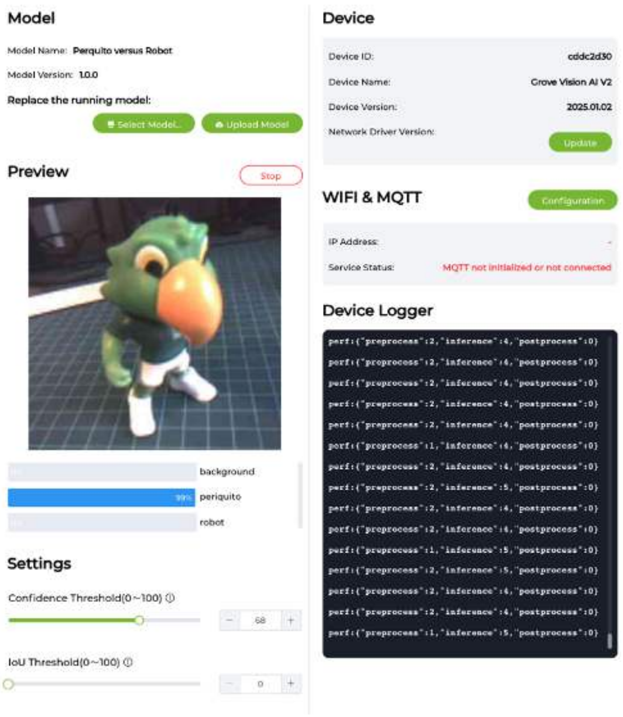
2025.01.02

MQTT not initialized or not connected

```
}
boxes:[]
perf:{"preprocess":4,"inference":48,"postprocess":1
}
boxes:[]
perf:{"preprocess":4,"inference":48,"postprocess":1
}
boxes:[]
perf:{"preprocess":4,"inference":48,"postprocess":1
}
boxes:[]
perf:{"preprocess":4,"inference":48,"postprocess":1
}
boxes:[]
```

After a few seconds, the model will be uploaded ("flashed") to our device, and the camera image will appear in real-time on the **Preview** Sector. The Classification result will be displayed under the image preview. It is also possible to select the `Confidence Threshold` of your inference using the cursor on **Settings**.

On the **Device Logger**, we can view the Serial Monitor, where we can observe the latency, which is approximately 1 to 2 ms for pre-processing and 4 to 5 ms for inference, aligning with the estimates made in Edge Impulse Studio.



Model

Model Name: Perquito versus Robot

Model Version: 1.0.0

Replace the running model:

Preview

Settings

Confidence Threshold(0~100): 68

IoU Threshold(0~100): 0

Device

Device ID: cddc2d30

Device Name: Grove Vision AI V2

Device Version: 2025.01.02

Network Driver Version:

WIFI & MQTT

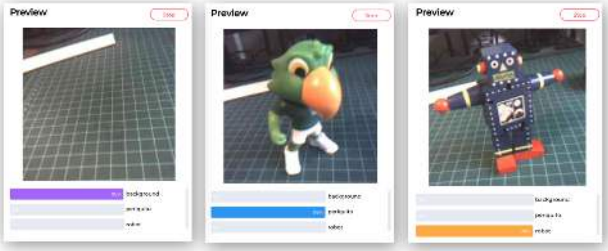
IP Address: -

Service Status: MQTT not initialized or not connected

Device Logger

```
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":1,"inference":4,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":2,"inference":5,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":1,"inference":5,"postprocess":0}
perf:{"preprocess":2,"inference":5,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":2,"inference":4,"postprocess":0}
perf:{"preprocess":1,"inference":5,"postprocess":0}
```

Here are other screenshots:

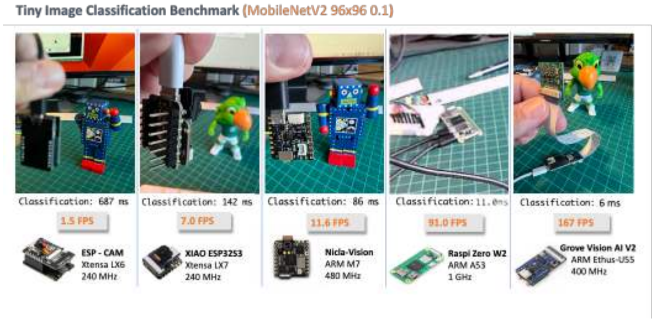


The power consumption of this model is approximately 70 mA, equivalent to 0.4 W.

Image Classification (non-official) Benchmark

Several development boards can be used for embedded machine learning (tinyML), and the most common ones (so far) for Computer Vision applications (with low energy) are the **ESP32 CAM,** the **Seeed XIAO ESP32S3 Sense**, and the Arduino **Nicla Vision**.

Taking advantage of this opportunity, a similarly trained model, MobilenetV2 96x96, with an alpha of 0.1, was also deployed on the ESP-CAM, the XIAO, and a Raspberry Pi Zero W2. Here is the result:



> The Grove Vision AI V2 with an **ARM Ethus-U55** was approximately 14 times faster than devices with an ARM-M7, and more than 100 times faster than an Xtensa LX6 (ESP-CAM). Even when compared to a Raspberry Pi, with a much more powerful CPU, the U55 reduces latency by almost half. Additionally, the power consumption is lower than that of other devices (see the full article here for power consumption comparison).

Postprocessing

Now that we have the model uploaded to the board and working correctly, classifying our images, let's connect a Master Device to export the inference result to it and see the result completely offline (disconnected from the PC and, for example, powered by a battery).

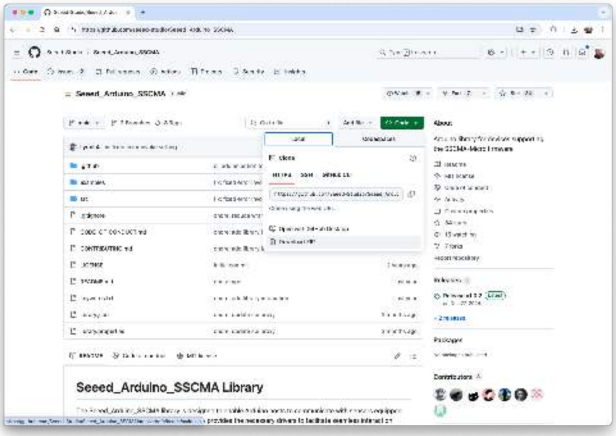
> Note that we can use any microcontroller as a Master Controller, such as the XIAO, Arduino, or Raspberry Pi.

Getting the Video Stream

The image processing and model inference are processed locally in Grove Vision AI (V2), and we want the result to be output to the XIAO (Master Controller) via IIC. For that, we will use the **Arduino SSMA library**. This library's primary purpose is to process Grove Vision AI's data stream, which does not involve model inference.

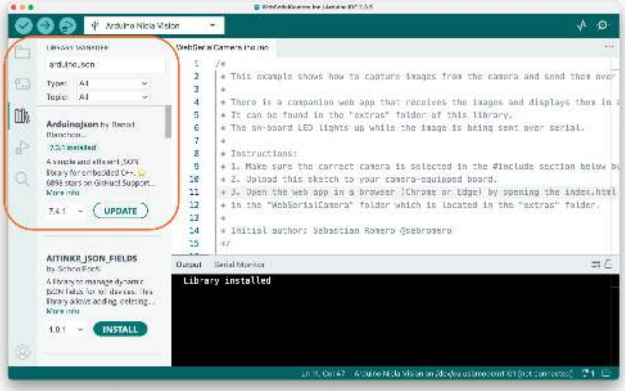
> The Grove Vision AI (V2) communicates (Inference result) with the XIAO via the IIC; the device's IIC address is 0x62. Image information transfer is via the USB serial port.

Step 1: Download the Arduino SSMA library as a zip file from its GitHub:



Step 2: Install it in the Arduino IDE (`sketch > Include Library > Add .Zip Library`).

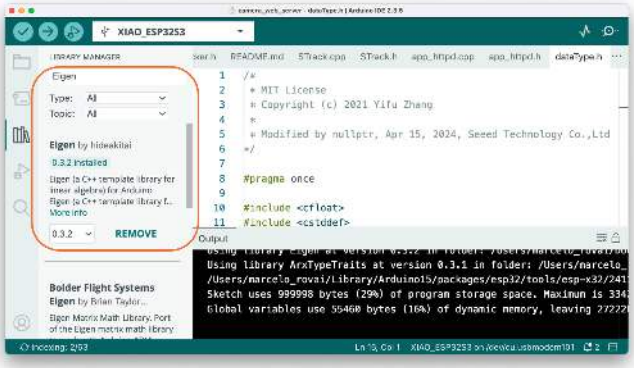
Step 3: Install the **ArduinoJSON** library.


Arduino Nicla Vision
LIBRARY MANAGER
arduinojson
Type: All
Topic: All
ArduinoJson by Benoit Blanchon
7.2.1 installed
A simple and efficient JSON library for embedded C++. 6898 stars on GitHub! Support...
More info
7.4.1
UPDATE
AITINKR_JSON_FIELDS
by Schoo FoeA
A library to manage dynamic JSON fields for IoT devices. This library allows adding, deleting...
More info
1.0.1
INSTALL
WebSerialCamera.ino
Output
Serial Monitor

```
/*
 * This example shows how to capture images from the camera and send them over
 *
 * There is a companion web app that receives the images and displays them in a
 * It can be found in the "extras" folder of this library.
 * The on-board LED lights up while the image is being sent over serial.
 *
 * Instructions:
 * 1. Make sure the correct camera is selected in the #include section below by
 * 2. Upload this sketch to your camera-equipped board.
 * 3. Open the web app in a browser (Chrome or Edge) by opening the index.html
 * in the "WebSerialCamera" folder which is located in the "extras" folder.
 *
 * Initial author: Sebastian Romero @sebromero
 */
```

```
Library installed
```

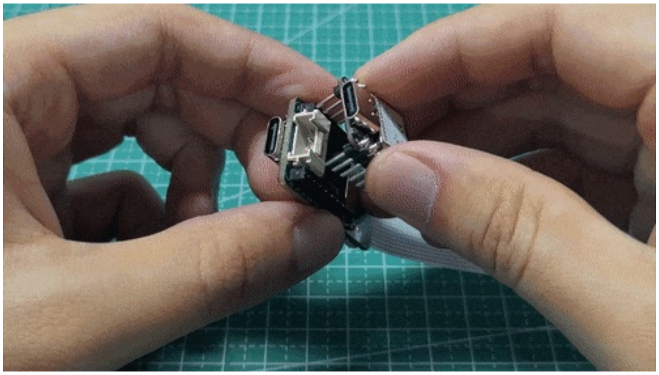
Step 4: Install the **Eigen** Library


camera_web_server - dataType.h | Arduino IDE 2.3.6
XIAO_ESP32S3
LIBRARY MANAGER
ser.h README.md STrack.cpp STrack.h app_httpd.cpp app_httpd.h dataType.h
Eigen
Type: All
Topic: All
Eigen by hideakitai
0.3.2 installed
Eigen (a C++ template library for linear algebra) for Arduino
Eigen (a C++ template library f...
More info
0.3.2
REMOVE
Bolder Flight Systems Eigen by Brian Taylor...
Eigen Matrix Math Library. Port of the Eigen matrix math library
Output
Indexing: 2/63
Ln 10, Col 1 XIAO_ESP32S3 on /dev/cu.usbmodem101 2

```
/*
 * MIT License
 * Copyright (c) 2021 Yifu Zhang
 *
 * Modified by nullptr, Apr 15, 2024, Seeed Technology Co.,Ltd
 */

#pragma once

#include <cfloat>
#include <cstddef>
```

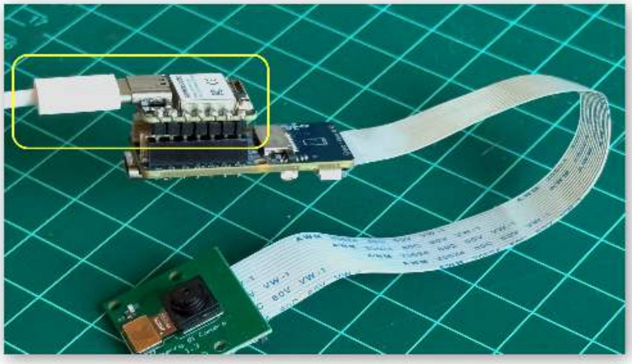
```
Using library Eigen at version 0.3.2 in folder: /Users/marcelo_rovai/Do
Using library ArxTypeTraits at version 0.3.1 in folder: /Users/marcelo_
/Users/marcelo_rovai/Library/Arduino15/packages/esp32/tools/esp-x32/241
Sketch uses 999998 bytes (29%) of program storage space. Maximum is 334
Global variables use 55460 bytes (16%) of dynamic memory, leaving 27222
```

Step 3: Now, connect the XIAO and Grove Vision AI (V2) via the socket (a row of pins) located at the back of the device.



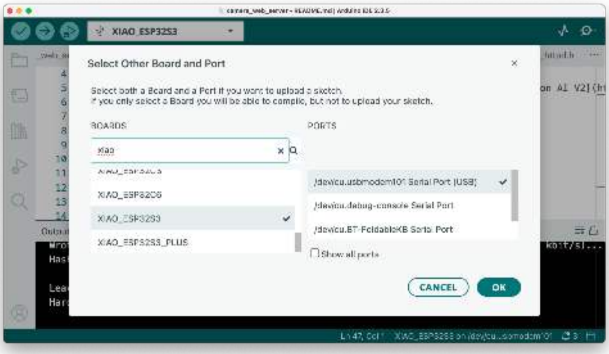
> **CAUTION**: Please note the direction of the connection, Grove Vision AI's Type-C connector should be in the same direction as XIAO's Type-C connector.

Step 5: Connect the **XIAO USB-C** port to your computer



Step 6: In the Arduino IDE, select the Xiao board and the corresponding USB port.

Once we want to stream the video to a webpage, we will use the **XIAO ESP32S3**, which has wifi and enough memory to handle images. Select `XIAO_ESP32S3` and the appropriate USB Port:



Select Other Board and Port

Select both a Board and a Port if you want to upload a sketch.
If you only select a Board you will be able to compile, but not to upload your sketch.

BOARDS: xiao

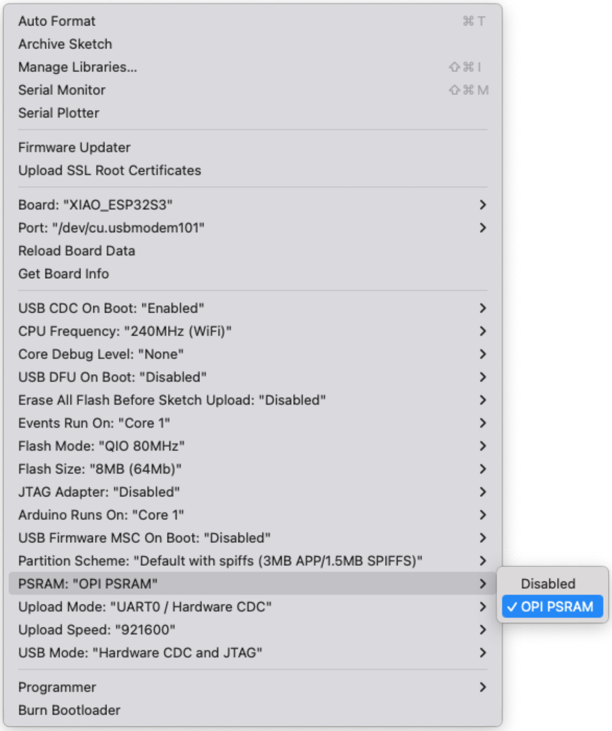
- XIAO_ESP8266
- XIAO_ESP32S3 ✓
- XIAO_ESP32S3_PLUS

PORTS

- /dev/cu.usbmodem101 Serial Port (USB) ✓
- /dev/cu.debug-console Serial Port
- /dev/cu.BT-FoldableKB Serial Port

□ Show all ports

By default, the PSRAM is disabled. Open the `Tools` menu and on PSRAM: `"OPI PSRAM"`select `OPI PSRAM`.



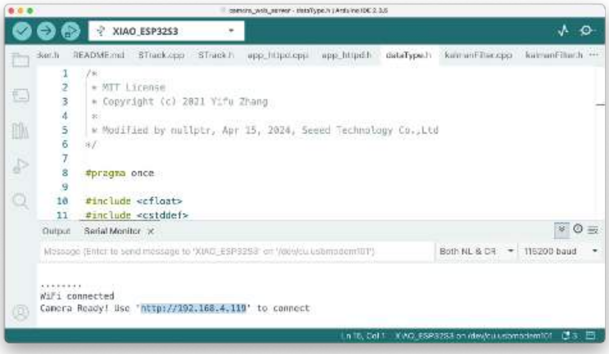
Board: "XIAO_ESP32S3"
Port: "/dev/cu.usbmodem101"
USB CDC On Boot: "Enabled"
CPU Frequency: "240MHz (WiFi)"
Core Debug Level: "None"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Events Run On: "Core 1"
Flash Mode: "QIO 80MHz"
Flash Size: "8MB (64Mb)"
JTAG Adapter: "Disabled"
Arduino Runs On: "Core 1"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "Default with spiffs (3MB APP/1.5MB SPIFFS)"
PSRAM: "OPI PSRAM"
✓ OPI PSRAM
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"

Step 7: Open the example in Arduino IDE:

`File -> Examples -> Seeed_Arduino_SSCMA -> camera_web_server`.

And edit the `ssid` and `password` in the `camera_web_server.ino` sketch to match the Wi-Fi network.

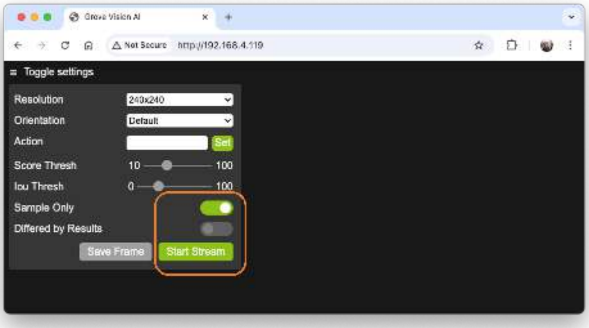
Step 8: Upload the sketch to the board and open the Serial Monitor. When connected to the Wi-Fi network, the board's IP address will be displayed.



```
/*
 * MIT License
 * Copyright (c) 2021 Yifu Zhang
 *
 * Modified by nullptr, Apr 15, 2024, Seeed Technology Co.,Ltd
*/

#pragma once

#include <cfloat>
#include <cstddef>
```

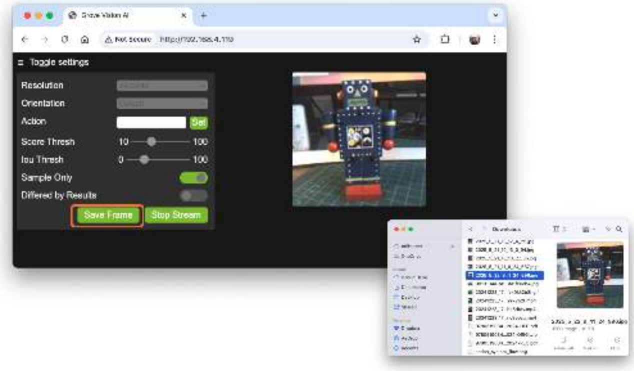
```
........
WiFi connected
Camera Ready! Use 'http://192.168.4.119' to connect
```

Open the address using a web browser. A Video App will be available. To see **only** the video stream from the Grove Vision AI V2, press [Sample Only] and [Start Stream].



Resolution 240x240
Orientation Default
Action
Score Thresh 10 — 100
Iou Thresh 0 — 100
Sample Only
Differed by Results

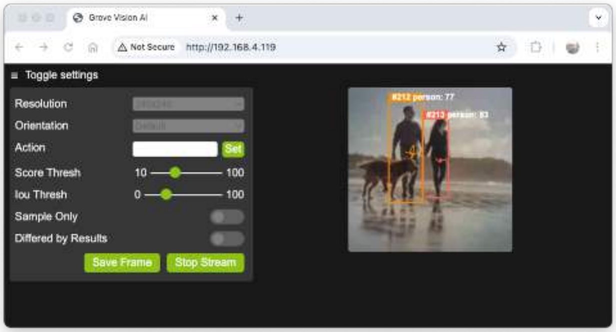
If you want to create an image dataset, you can use this app, saving frames of the video generated by the device. Pressing [Save Frame], the image will be saved in the download area of our desktop.



Resolution
Orientation
Action
Score Thresh 10 — 100
Iou Thresh 0 — 100
Sample Only
Differed by Results

Opening the App **without** selecting [Sample Only], the inference result should appear on the video screen, but this does not happen for Image Classification. For Object Detection or Pose Estimation, the result is embedded with the video stream.

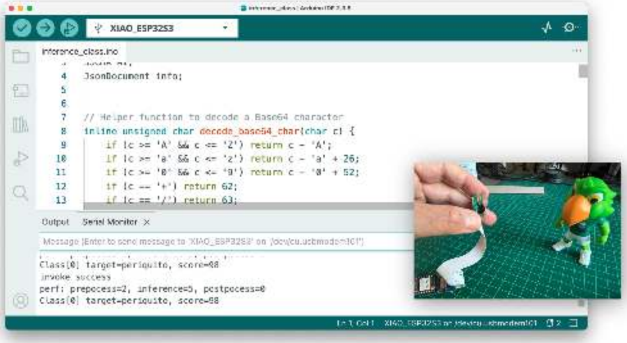
For example, if the model is a Person Detection using YoloV8:



Resolution
Orientation
Action Set
Score Thresh 10 100
Iou Thresh 0 100
Sample Only
Differed by Results

Getting the Inference Result

- Go to `File -> Examples -> Seeed_Arduino_SSCMA -> inference_class`.
- Upload the sketch to the board, and open the Serial Monitor.
- Pointing the camera at one of our objects, we can see the inference result on the Serial Terminal.



XIAO_ESP32S3
inference_class.ino
Output
Serial Monitor
Message (Enter to send message to 'XIAO_ESP32S3' on '/dev/cu.usbmodem101')

```
JsonDocument info;


// Helper function to decode a Base64 character
inline unsigned char decode_base64_char(char c) {
    if (c >= 'A' && c <= 'Z') return c - 'A';
    if (c >= 'a' && c <= 'z') return c - 'a' + 26;
    if (c >= '0' && c <= '9') return c - '0' + 52;
    if (c == '+') return 62;
    if (c == '/') return 63;
```

```
Class[0] target=periquito, score=88
invoke success
perf: prepocess=2, inference=5, postpocess=0
Class[0] target=periquito, score=88
```

The inference running on the Arduino IDE had an average consumption of 160 mA or 800 mW and a peak of 330 mA 1.65 W when transmitting the image to the App.

Postprocessing with LED

The idea behind our postprocessing is that whenever a specific image is detected (for example, the Periquito - Label:1), the User LED is turned on. If the Robot or a background is detected, the LED will be off.

Copy the below code and past it to your IDE:

```
#include <Seeed_Arduino_SSCMA.h>
SSCMA AI;
```

```
void setup()
{
    AI.begin();

    Serial.begin(115200);
    while (!Serial);
    Serial.println("Inferencing - Grove AI V2 / XIAO ESP32S3");

    // Pins for the built-in LED
    pinMode(LED_BUILTIN, OUTPUT);
    // Ensure the LED is OFF by default.
    // Note: The LED is ON when the pin is LOW, OFF when HIGH.
    digitalWrite(LED_BUILTIN, HIGH);
}

void loop()
{
    if (!AI.invoke()){
        Serial.println("\nInvoke Success");
        Serial.print("Latency [ms]: prepocess=");
        Serial.print(AI.perf().prepocess);
        Serial.print(", inference=");
        Serial.print(AI.perf().inference);
        Serial.print(", postpocess=");
        Serial.println(AI.perf().postprocess);
        int pred_index = AI.classes()[0].target;
        Serial.print("Result= Label: ");
        Serial.print(pred_index);
        Serial.print(", score=");
        Serial.println(AI.classes()[0].score);
        turn_on_led(pred_index);
    }
}

/**
* @brief      turn_off_led function - turn-off the User LED
*/
void turn_off_led(){
    digitalWrite(LED_BUILTIN, HIGH);
}

/**
* @brief      turn_on_led function used to turn on the User LED
* @param[in]  pred_index
*             label 0: [0] ==> ALL OFF
*             label 1: [1] ==> LED ON
*             label 2: [2] ==> ALL OFF
*              label 3: [3] ==> ALL OFF
*/
void turn_on_led(int pred_index) {
    switch (pred_index)
    {
        case 0:
            turn_off_led();
            break;
        case 1:
            turn_off_led();
            digitalWrite(LED_BUILTIN, LOW);
            break;
        case 2:
            turn_off_led();
            break;
        case 3:
            turn_off_led();
            break;
    }
}
```

This sketch uses the Seeed_Arduino_SSCMA.h library to interface with the Grove Vision AI Module V2. The AI module and the LED are initialized in the `setup()` function, and serial communication is started.

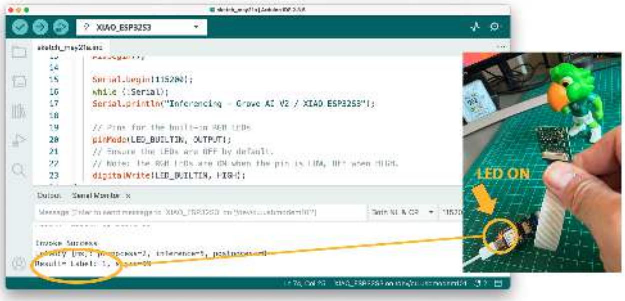
The `loop()` function repeatedly calls the `invoke()` method to perform inference using the built-in algorithms of the Grove Vision AI Module V2. Upon a successful inference, the sketch prints out performance metrics to the serial monitor, including preprocessing, inference, and postprocessing times.

The sketch processes and prints out detailed information about the results of the inference:

- (`AI.classes()[0]`) that identifies the class of image (`.target`) and its confidence score (`.score`).
- The inference result (class) is stored in the integer variable `pred_index`, which will be used as an input to the function `turn_on_led()`. As a result, the LED will turn ON, depending on the classification result.

Here is the result:

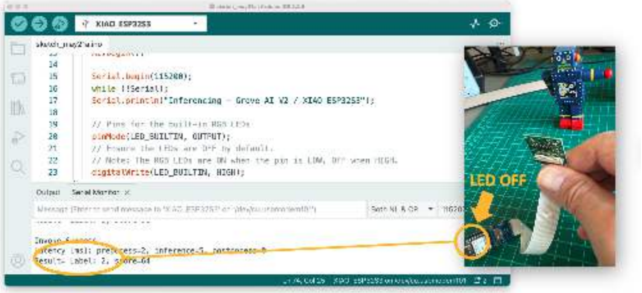
If the Periquito is detected (Label:1), the LED is ON:



```
    Serial.begin(115200);
    while (!Serial);
    Serial.println("Inferencing - Grove AI V2 / XIAO ESP32S3");

    // Pins for the built-in RGB LEDs
    pinMode(LED_BUILTIN, OUTPUT);
    // Ensure the LEDs are OFF by default.
    // Note: The RGB LEDs are ON when the pin is LOW, OFF when HIGH.
    digitalWrite(LED_BUILTIN, HIGH);
```

```
Invoke Success
Latency [ms]: preprocess=2, inference=5, postprocess=0
Result: Label: 1, score=0
```

If the Robot is detected (Label:2) the LED is OFF (Same for Background (Label:0):



```
    Serial.begin(115200);
    while (!Serial);
    Serial.println("Inferencing - Grove AI V2 / XIAO ESP32S3");

    // Pins for the built-in RGB LEDs
    pinMode(LED_BUILTIN, OUTPUT);
    // Ensure the LEDs are OFF by default.
    // Note: The RGB LEDs are ON when the pin is LOW, OFF when HIGH.
    digitalWrite(LED_BUILTIN, HIGH);
```

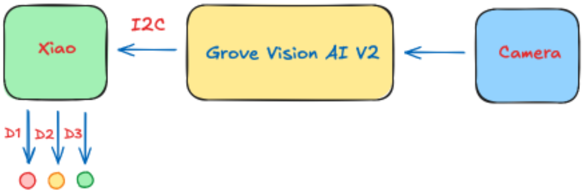
```
Invoke Success
Latency [ms]: preprocess=2, inference=5, postprocess=0
Result: Label: 2, score=64
```

Therefore, we can now power the Grove Viaon AI V2 + Xiao ESP32S3 with an external battery, and the inference result will be displayed by the LED completely offline. The consumption is approximately 165 mA or 825 mW.

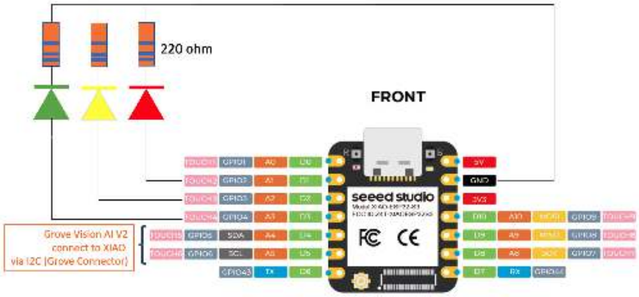
> It is also possible to send the result using Wifi, BLE, or other communication protocols available on the used Master Device.

Optional: Post-processing on external devices

Of course, one of the significant advantages of working with EdgeAI is that devices can run entirely disconnected from the cloud, allowing for seamless **interactions with the real world**. We did it in the last section, but using the internal Xiao LED. Now, we will connect external LEDs (which could be any actuator).



The LEDS should be connected to the XIAO ground via a 220-ohm resistor.



The idea is to modify the previous sketch to handle the three external LEDs.

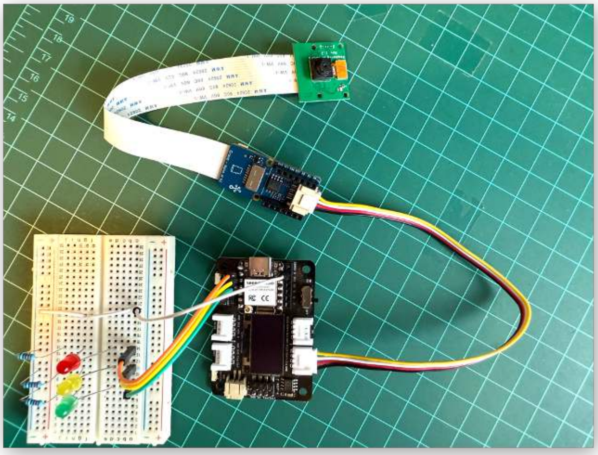
GOAL: Whenever the image of a **Periquito** is detected, the LED **Green** will be ON; if it is a **Robot**, the LED **Yellow** will be ON; if it is a **Background**, the **LED Red** will be ON.

The image processing and model inference are processed locally in Grove Vision AI (V2), and we want the result to be output to the XIAO via IIC. For that, we will use the Arduino SSMA library again.

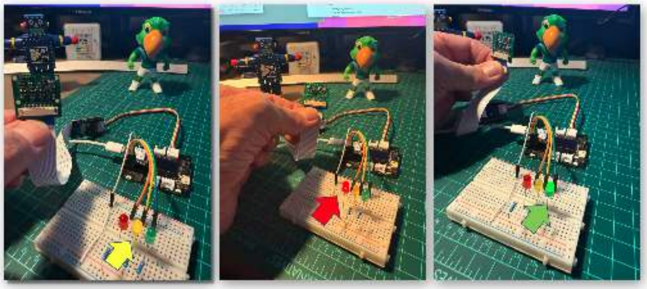
Here the sketch to be used:

```
#include <Seeed_Arduino_SSCMA.h>
SSCMA AI;

// Define the LED pin according to the pin diagram
// The LEDS negative lead should be connected to the XIAO ground
// via a 220-ohm resistor.
int LEDR = D1; # XIAO ESP32S3 Pin 1
int LEDY = D2; # XIAO ESP32S3 Pin 2
int LEDG = D3; # XIAO ESP32S3 Pin 3

  void setup()
{
    AI.begin();
    Serial.begin(115200);
    while (!Serial);
    Serial.println("Inferencing - Grove AI V2 / XIAO ESP32S3");

// Initialize the external LEDs
    pinMode(LEDR, OUTPUT);
    pinMode(LEDY, OUTPUT);
    pinMode(LEDG, OUTPUT);
    // Ensure the LEDs are OFF by default.
    // Note: The LEDs are ON when the pin is HIGH, OFF when LOW.
    digitalWrite(LEDR, LOW);
    digitalWrite(LEDY, LOW);
    digitalWrite(LEDG, LOW);
}

void loop()
{
    if (!AI.invoke()){
        Serial.println("\nInvoke Success");
        Serial.print("Latency [ms]: prepocess=");
        Serial.print(AI.perf().prepocess);
        Serial.print(", inference=");
        Serial.print(AI.perf().inference);
        Serial.print(", postpocess=");
        Serial.println(AI.perf().postprocess);
        int pred_index = AI.classes()[0].target;
        Serial.print("Result= Label: ");
        Serial.print(pred_index);
        Serial.print(", score=");
        Serial.println(AI.classes()[0].score);
        turn_on_leds(pred_index);
    }
}


/**
* @brief turn_off_leds function - turn-off all LEDs
*/
void turn_off_leds(){
    digitalWrite(LEDR, LOW);
    digitalWrite(LEDY, LOW);
    digitalWrite(LEDG, LOW);
}

/**
* @brief turn_on_leds function used to turn on a specific LED
* @param[in]  pred_index
*              label 0: [0] ==> Red ON
*              label 1: [1] ==> Green ON
*              label 2: [2] ==> Yellow ON
*/
void turn_on_leds(int pred_index) {
    switch (pred_index)
    {
        case 0:
            turn_off_leds();
            digitalWrite(LEDR, HIGH);
            break;
        case 1:
            turn_off_leds();
            digitalWrite(LEDG, HIGH);
            break;
        case 2:
            turn_off_leds();
            digitalWrite(LEDY, HIGH);
            break;
        case 3:
            turn_off_leds();
            break;
    }
}
```

We should connect the Grove Vision AI V2 with the XIAO using its I2C Grove connector. For the XIAO, we will use an Expansion Board for the facility (although it is possible to connect the I2C directly to the XIAO's pins). We will power the boards using the USB-C connector, but a battery can also be used.



Here is the result:



> The power consumption reached a peak of 240 mA (Green LED), equivalent to 1.2 W. Driving the Yellow and Red LEDs consumes 14 mA, equivalent to 0.7 W. Sending information to the terminal via serial has no impact on power consumption.

Conclusion

In this lab, we've explored the complete process of developing an image classification system using the Seeed Studio Grove Vision AI Module V2 powered by the Himax WiseEye2 chip. We've walked through every stage of the machine learning workflow, from defining our project goals to deploying a working model with real-world interactions.

The Grove Vision AI V2 has demonstrated impressive performance, with inference times of just 4-5ms, dramatically outperforming other common tinyML platforms. Our benchmark comparison showed it to be approximately 14 times faster than ARM-M7 devices and over 100 times faster than an Xtensa LX6 (ESP-CAM). Even when compared to a Raspberry Pi Zero W2, the Edge TPU architecture delivered nearly twice the speed while consuming less power.

Through this project, we've seen how transfer learning enables us to achieve good classification results with a relatively small dataset of custom images. The MobileNetV2 model with an alpha of 0.1 provided an excellent balance of accuracy and efficiency for our three-class problem, requiring only 146 KB of RAM and 187 KB of Flash memory, well within the capabilities of the Grove Vision AI Module V2's 2.4 MB internal SRAM.

We also explored several deployment options, from viewing inference results through the SenseCraft AI Studio to creating a standalone system with visual feedback using LEDs. The ability to stream video to a web browser and process inference results locally demonstrates the versatility of edge AI systems for real-world applications.

The power consumption of our final system remained impressively low, ranging from approximately 70mA (0.4W) for basic inference to 240mA (1.2W) when driving external components. This efficiency makes the Grove Vision AI Module V2 an excellent choice for battery-powered applications where power consumption is critical.

This lab has demonstrated that sophisticated computer vision tasks can now be performed entirely at the edge, without reliance on cloud services or powerful computers. With tools like Edge Impulse Studio and SenseCraft AI Studio, the development process has become accessible even to those without extensive machine learning expertise.

As edge AI technology continues to evolve, we can expect even more powerful capabilities from compact, energy-efficient devices like the Grove Vision AI Module V2, opening up new possibilities for smart sensors, IoT applications, and embedded intelligence in everyday objects.

Resources

Collecting Images with SenseCraft AI Studio.

Edge Impulse Studio Project

SenseCraft AI Studio - Vision Workplace (Deploy Models)

Other Himax examples

Arduino Sketches

Object Detection

This Lab is under Development

Raspberry Pi

These labs offer invaluable hands-on experience with machine learning systems, leveraging the versatility and accessibility of the Raspberry Pi platform. Unlike working with large-scale models that demand extensive cloud resources, these exercises allow you to directly interact with hardware and software in a compact yet powerful edge computing environment. You'll gain practical insights into deploying AI at the edge by utilizing Raspberry Pi's capabilities, from the efficient Pi Zero to the more robust Pi 4 or Pi 5 models. This approach provides a tangible understanding of the challenges and opportunities in implementing machine learning solutions in resource-constrained settings. While we're working at a smaller scale, the principles and techniques you'll learn are fundamentally similar to those used in larger systems. The Raspberry Pi's ability to run a whole operating system and its extensive GPIO capabilities allow for a rich learning experience that bridges the gap between theoretical knowledge and real-world application. Through these labs, you'll grasp the intricacies of EdgeML and develop skills applicable to a wide range of AI deployment scenarios.

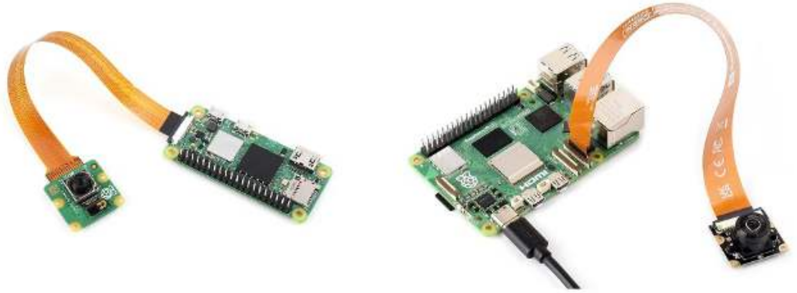


Figure 20.15: Raspberry Pi Zero 2-W and Raspberry Pi 5 with Camera

Pre-requisites

- **Raspberry Pi**: Ensure you have at least one of the boards: the Raspberry Pi Zero 2 W, Raspberry Pi 4 or 5 for the Vision Labs, and the Raspberry 5 for the GenAi labs.

- **Power Adapter**: To Power on the boards.
  - Raspberry Pi Zero 2-W: 2.5 W with a Micro-USB adapter
  - Raspberry Pi 4 or 5: 3.5 W with a USB-C adapter
- **Network**: With internet access for downloading the necessary software and controlling the boards remotely.
- **SD Card (32 GB minimum) and an SD card Adapter**: For the Raspberry Pi OS.

Setup

- Setup Raspberry Pi

Exercises

Modality	Task	Description	Link
Vision	Image Classification	Learn to classify images	Link
Vision	Object Detection	Implement object detection	Link
GenAI	Small Language Models	Deploy SLMs at the Edge	Link
GenAI	Visual-Language Models	Deploy VLMs at the Edge	Link

Setup

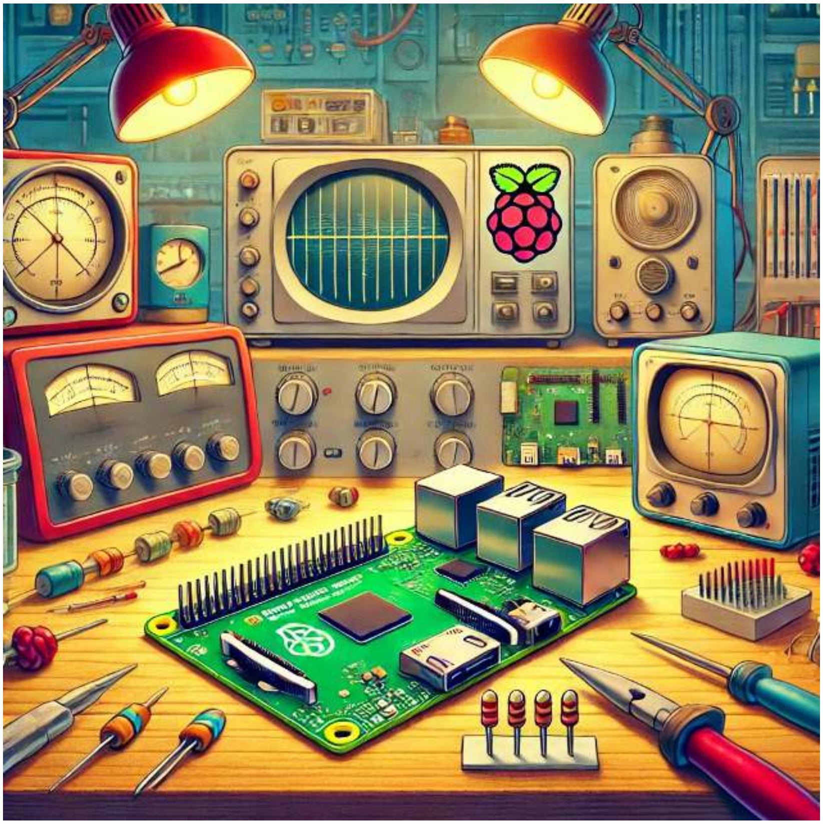


Figure 20.16: *DALL·E prompt - An electronics laboratory environment inspired by the 1950s, with a cartoon style. The lab should have vintage equipment, large oscilloscopes, old-fashioned tube radios, and large, boxy computers. The Raspberry Pi 5 board is prominently displayed, accurately shown in its real size, similar to a credit card, on a workbench. The Pi board is surrounded by classic lab tools like a soldering iron, resistors, and wires. The overall scene should be vibrant, with exaggerated colors and playful details characteristic of a cartoon. No logos or text should be included.*

This chapter will guide you through setting up Raspberry Pi Zero 2 W (*Raspi-Zero*) and Raspberry Pi 5 (*Raspi-5*) models. We'll cover hardware setup, operating system installation, initial configuration, and tests.

> The general instructions for the *Raspi-5* also apply to the older Raspberry Pi versions, such as the Raspi-3 and Raspi-4.

Overview

The Raspberry Pi is a powerful and versatile single-board computer that has become an essential tool for engineers across various disciplines. Developed by the Raspberry Pi Foundation, these compact devices offer a unique combination of affordability, computational power, and extensive GPIO (General Purpose Input/Output) capabilities, making them ideal for prototyping, embedded systems development, and advanced engineering projects.

Key Features

1. **Computational Power**: Despite their small size, Raspberry Pis offers significant processing capabilities, with the latest models featuring multi-core ARM processors and up to 8 GB of RAM.
2. **GPIO Interface**: The 40-pin GPIO header allows direct interaction with sensors, actuators, and other electronic components, facilitating hardware-software integration projects.
3. **Extensive Connectivity**: Built-in Wi-Fi, Bluetooth, Ethernet, and multiple USB ports enable diverse communication and networking projects.
4. **Low-Level Hardware Access**: Raspberry Pis provides access to interfaces like I2C, SPI, and UART, allowing for detailed control and communication with external devices.
5. **Real-Time Capabilities**: With proper configuration, Raspberry Pis can be used for soft real-time applications, making them suitable for control systems and signal processing tasks.
6. **Power Efficiency**: Low power consumption enables battery-powered and energy-efficient designs, especially in models like the Pi Zero.

Raspberry Pi Models (covered in this book)

1. **Raspberry Pi Zero 2 W** (*Raspi-Zero*):
   - Ideal for: Compact embedded systems
   - Key specs: 1 GHz single-core CPU (ARM Cortex-A53), 512 MB RAM, minimal power consumption
2. **Raspberry Pi 5** (*Raspi-5*):
   - Ideal for: More demanding applications such as edge computing, computer vision, and edgeAI applications, including LLMs.
   - Key specs: 2.4 GHz quad-core CPU (ARM Cortex A-76), up to 8 GB RAM, PCIe interface for expansions

Engineering Applications

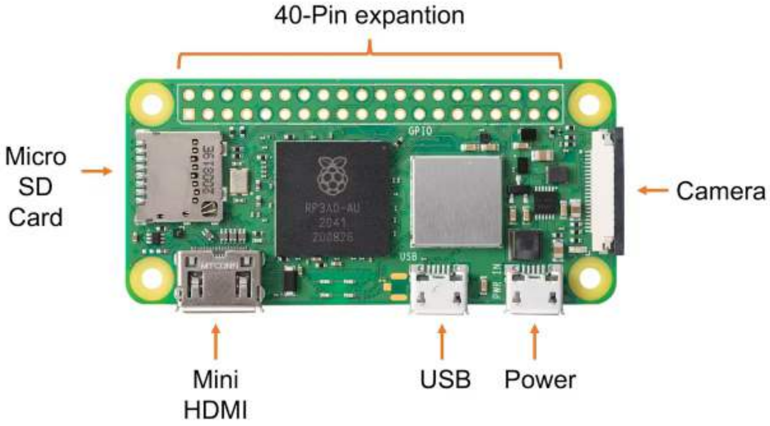
1. **Embedded Systems Design**: Develop and prototype embedded systems for real-world applications.
2. **IoT and Networked Devices**: Create interconnected devices and explore protocols like MQTT, CoAP, and HTTP/HTTPS.

3. **Control Systems**: Implement feedback control loops, PID controllers, and interface with actuators.
4. **Computer Vision and AI**: Utilize libraries like OpenCV and TensorFlow Lite for image processing and machine learning at the edge.
5. **Data Acquisition and Analysis**: Collect sensor data, perform real-time analysis, and create data logging systems.
6. **Robotics**: Build robot controllers, implement motion planning algorithms, and interface with motor drivers.
7. **Signal Processing**: Perform real-time signal analysis, filtering, and DSP applications.
8. **Network Security**: Set up VPNs, firewalls, and explore network penetration testing.

This tutorial will guide you through setting up the most common Raspberry Pi models, enabling you to start on your machine learning project quickly. We'll cover hardware setup, operating system installation, and initial configuration, focusing on preparing your Pi for Machine Learning applications.

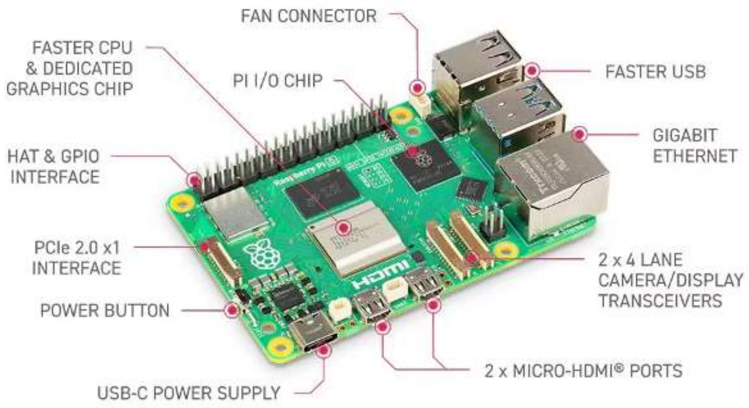
Hardware Overview

Raspberry Pi Zero 2W



- **Processor**: 1 GHz quad-core 64-bit Arm Cortex-A53 CPU
- **RAM**: 512 MB SDRAM
- **Wireless**: 2.4 GHz 802.11 b/g/n wireless LAN, Bluetooth 4.2, BLE
- **Ports**: Mini HDMI, micro USB OTG, CSI-2 camera connector
- **Power**: 5 V via micro USB port

Raspberry Pi 5



- **Processor**:
  - Pi 5: Quad-core 64-bit Arm Cortex-A76 CPU @ 2.4 GHz
  - Pi 4: Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5 GHz
- **RAM**: 2 GB, 4 GB, or 8 GB options (8 GB recommended for AI tasks)
- **Wireless**: Dual-band 802.11ac wireless, Bluetooth 5.0
- **Ports**: 2 × micro HDMI ports, 2 × USB 3.0 ports, 2 × USB 2.0 ports, CSI camera port, DSI display port
- **Power**: 5 V DC via USB-C connector (3A)

In the labs, we will use different names to address the Raspberry: `Raspi`, `Raspi-5`, `Raspi-Zero`, etc. Usually, `Raspi` is used when the instructions or comments apply to every model.

Installing the Operating System

The Operating System (OS)

An operating system (OS) is fundamental software that manages computer hardware and software resources, providing standard services for computer programs. It is the core software that runs on a computer, acting as an intermediary between hardware and application software. The OS manages the computer's memory, processes, device drivers, files, and security protocols.

1. **Key functions**:
   - Process management: Allocating CPU time to different programs
   - Memory management: Allocating and freeing up memory as needed
   - File system management: Organizing and keeping track of files and directories

- Device management: Communicating with connected hardware devices
- User interface: Providing a way for users to interact with the computer

2. **Components**:
   - Kernel: The core of the OS that manages hardware resources
   - Shell: The user interface for interacting with the OS
   - File system: Organizes and manages data storage
   - Device drivers: Software that allows the OS to communicate with hardware

The Raspberry Pi runs a specialized version of Linux designed for embedded systems. This operating system, typically a variant of Debian called Raspberry Pi OS (formerly Raspbian), is optimized for the Pi's ARM-based architecture and limited resources.

> The latest version of Raspberry Pi OS is based on Debian Bookworm.

Key features:

1. Lightweight: Tailored to run efficiently on the Pi's hardware.
2. Versatile: Supports a wide range of applications and programming languages.
3. Open-source: Allows for customization and community-driven improvements.
4. GPIO support: Enables interaction with sensors and other hardware through the Pi's pins.
5. Regular updates: Continuously improved for performance and security.

Embedded Linux on the Raspberry Pi provides a full-featured operating system in a compact package, making it ideal for projects ranging from simple IoT devices to more complex edge machine-learning applications. Its compatibility with standard Linux tools and libraries makes it a powerful platform for development and experimentation.

Installation

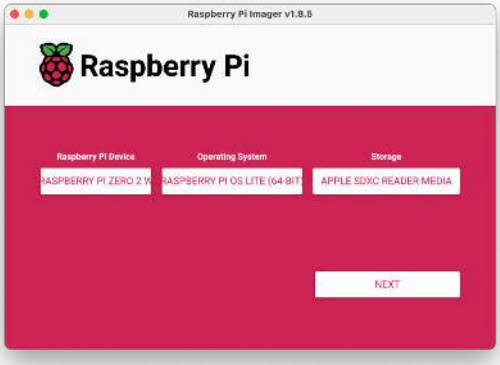
To use the Raspberry Pi, we will need an operating system. By default, Raspberry Pi checks for an operating system on any SD card inserted in the slot, so we should install an operating system using Raspberry Pi Imager.

Raspberry Pi Imager is a tool for downloading and writing images on *macOS*, *Windows*, and *Linux*. It includes many popular operating system images for Raspberry Pi. We will also use the Imager to preconfigure credentials and remote access settings.

Follow the steps to install the OS in your Raspi.

1. Download and install the Raspberry Pi Imager on your computer.
2. Insert a microSD card into your computer (a 32GB SD card is recommended) .

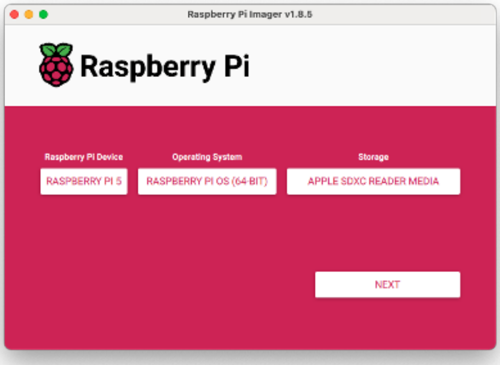
3. Open Raspberry Pi Imager and select your Raspberry Pi model.
4. Choose the appropriate operating system:
   - **For Raspi-Zero**: For example, you can select: `Raspberry Pi OS Lite (64-bit)`.



Raspberry Pi Device: ASPBERRY PI ZERO 2 W
Operating System: ASPBERRY PI OS LITE (64-BIT
Storage: APPLE SDXC READER MEDIA

   Due to its reduced SDRAM (512 MB), the recommended OS for the Raspi-Zero is the 32-bit version. However, to run some machine learning models, such as the YOLOv8 from Ultralitics, we should use the 64-bit version. Although Raspi-Zero can run a *desktop*, we will choose the LITE version (no Desktop) to reduce the RAM needed for regular operation.

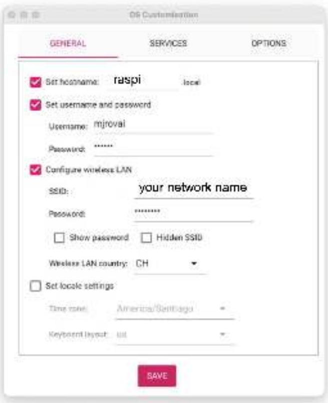
- For **Raspi-5**: We can select the full 64-bit version, which includes a desktop: `Raspberry Pi OS (64-bit)`



Raspberry Pi Device: RASPBERRY PI 5
Operating System: RASPBERRY PI OS (64-BIT)
Storage: APPLE SDXC READER MEDIA

5. Select your microSD card as the storage device.

6. Click on `Next` and then the `gear` icon to access advanced options.
7. Set the *hostname*, the Raspi *username and password*, configure *WiFi* and *enable SSH* (Very important!)



☑ Set hostname: raspi .local

☑ Set username and password

Username: mjrovai

Password: ******

☑ Configure wireless LAN

SSID: your network name

Password: ********

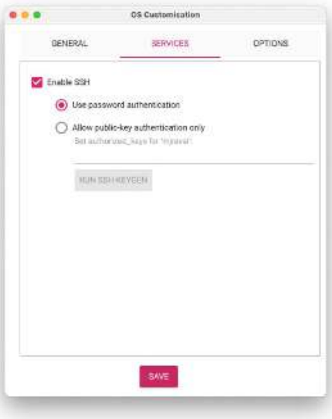
□ Show password □ Hidden SSID

Wireless LAN country: CH

□ Set locale settings

Time zone: America/Santiago

Keyboard layout: us



☑ Enable SSH

● Use password authentication

○ Allow public-key authentication only

Set authorized_keys for 'mjrovai':

8. Write the image to the microSD card.

In the examples here, we will use different hostnames depending on the device used: raspi, raspi-5, raspi-Zero, etc. It would help if you replaced it with the one you are using.

Initial Configuration

1. Insert the microSD card into your Raspberry Pi.
2. Connect power to boot up the Raspberry Pi.
3. Please wait for the initial boot process to complete (it may take a few minutes).

You can find the most common Linux commands to be used with the Raspi here or here.

Remote Access

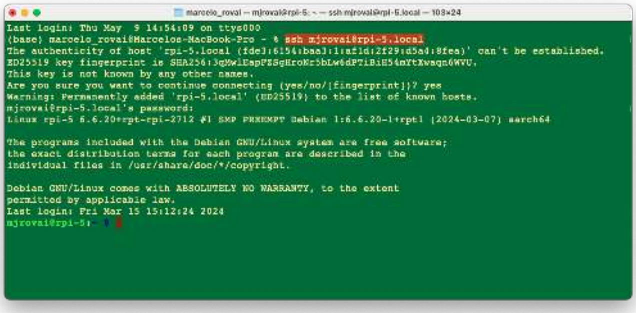
SSH Access

The easiest way to interact with the Raspi-Zero is via SSH ("Headless"). You can use a Terminal (MAC/Linux), PuTTy (Windows), or any other.

1. Find your Raspberry Pi's IP address (for example, check your router).
2. On your computer, open a terminal and connect via SSH:

```
ssh username@[raspberry_pi_ip_address]
```

Alternatively, if you do not have the IP address, you can try the following: `bash ssh username@hostname.local` for example, `ssh mjrovai@rpi-5.local` , `ssh mjrovai@raspi.local` , etc.


marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 103×24

```
Last login: Thu May  9 14:54:09 on ttys000
(base) marcelo_rovai@Marcelos-MacBook-Pro ~ % ssh mjrovai@rpi-5.local
The authenticity of host 'rpi-5.local (fde3:6154:baa3:1:af1d:2f29:d5a4:8fea)' can't be established.
ED25519 key fingerprint is SHA256:3qMwlEapFSSgHroKr5bLw6dPTiBiH54mYtXwaqn6WVU.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'rpi-5.local' (ED25519) to the list of known hosts.
mjrovai@rpi-5.local's password:
Linux rpi-5 6.6.20+rpt-rpi-2712 #1 SMP PREEMPT Debian 1:6.6.20-1+rpt1 (2024-03-07) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Mar 15 15:12:24 2024
mjrovai@rpi-5:~ $
```

Figure 20.17: img

When you see the prompt:

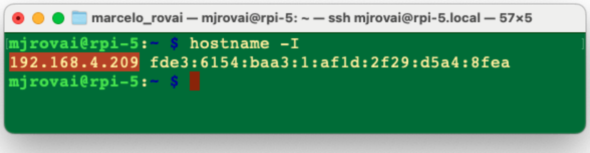
```
mjrovai@rpi-5:~ $
```

It means that you are interacting remotely with your Raspi. It is a good practice to update/upgrade the system regularly. For that, you should run:

```
sudo apt-get update
sudo apt upgrade
```

You should confirm the Raspi IP address. On the terminal, you can use:

```
hostname -I
```


marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 57×5

```
mjrovai@rpi-5:~ $ hostname -I
192.168.4.209 fde3:6154:baa3:1:af1d:2f29:d5a4:8fea
mjrovai@rpi-5:~ $
```

To shut down the Raspi via terminal:

When you want to turn off your Raspberry Pi, there are better ideas than just pulling the power cord. This is because the Raspi may still be writing data to the SD card, in which case merely powering down may result in data loss or, even worse, a corrupted SD card.

For safety shut down, use the command line:

```
sudo shutdown -h now
```

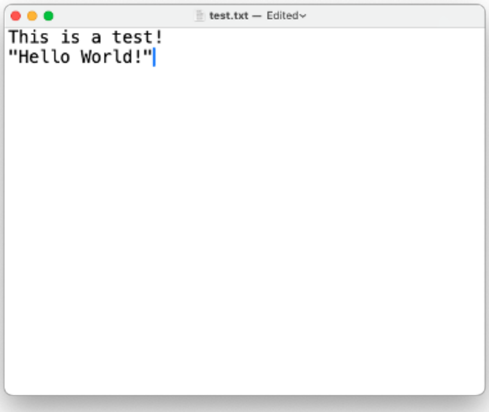
> To avoid possible data loss and SD card corruption, before removing the power, you should wait a few seconds after shutdown for the Raspberry Pi's LED to stop blinking and go dark. Once the LED goes out, it's safe to power down.

Transfer Files between the Raspi and a computer

Transferring files between the Raspi and our main computer can be done using a pen drive, directly on the terminal (with scp), or an FTP program over the network.

Using Secure Copy Protocol (scp):

Copy files to your Raspberry Pi. Let's create a text file on our computer, for example, `test.txt`.



```
This is a test!
"Hello World!"
```

> You can use any text editor. In the same terminal, an option is the `nano`.

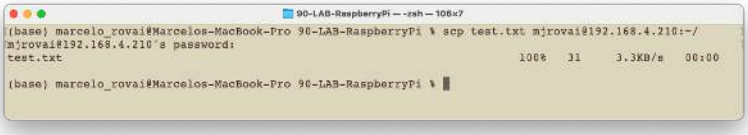
To copy the file named `test.txt` from your personal computer to a user's home folder on your Raspberry Pi, run the following command from the directory containing `test.txt`, replacing the `<username>` placeholder with the username you use to log in to your Raspberry Pi and the `<pi_ip_address>` placeholder with your Raspberry Pi's IP address:

```
$ scp test.txt <username>@<pi_ip_address>:~/
```

> Note that `~/` means that we will move the file to the ROOT of our Raspi. You can choose any folder in your Raspi. But you should create the folder before you run `scp`, since `scp` won't create folders automatically.

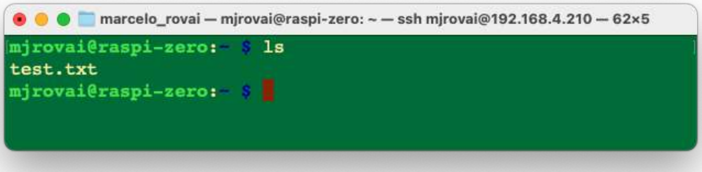
For example, let's transfer the file `test.txt` to the ROOT of my Raspi-zero, which has an IP of `192.168.4.210`:

```
scp test.txt mjrovai@192.168.4.210:~/
```



```
(base) marcelo_rovai@Marcelos-MacBook-Pro 90-LAB-RaspberryPi % scp test.txt mjrovai@192.168.4.210:~/
mjrovai@192.168.4.210's password:
test.txt                                                        100%   31     3.3KB/s   00:00

(base) marcelo_rovai@Marcelos-MacBook-Pro 90-LAB-RaspberryPi %
```

I use a different profile to differentiate the terminals. The above action happens **on your computer**. Now, let's go to our Raspi (using the SSH) and check if the file is there:



marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 62×5

```
mjrovai@raspi-zero:~ $ ls
test.txt
mjrovai@raspi-zero:~ $
```

Copy files from your Raspberry Pi. To copy a file named `test.txt` from a user's home directory on a Raspberry Pi to the current directory on another computer, run the following command **on your Host Computer**:

```
$ scp <username>@<pi_ip_address>:myfile.txt .
```

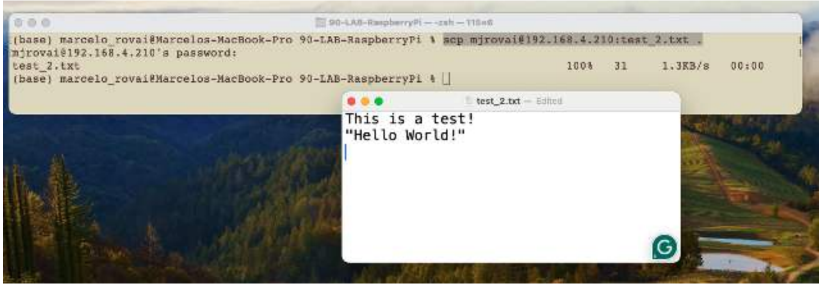
For example:

On the Raspi, let's create a copy of the file with another name:

```
cp test.txt test_2.txt
```

And on the Host Computer (in my case, a Mac)

```
scp mjrovai@192.168.4.210:test_2.txt .
```



```
(base) marcelo_rovai@Marcelos-MacBook-Pro 90-LAB-RaspberryPi % scp mjrovai@192.168.4.210:test_2.txt .
mjrovai@192.168.4.210's password:
test_2.txt                                                      100%   31     1.3KB/s   00:00
(base) marcelo_rovai@Marcelos-MacBook-Pro 90-LAB-RaspberryPi %
```

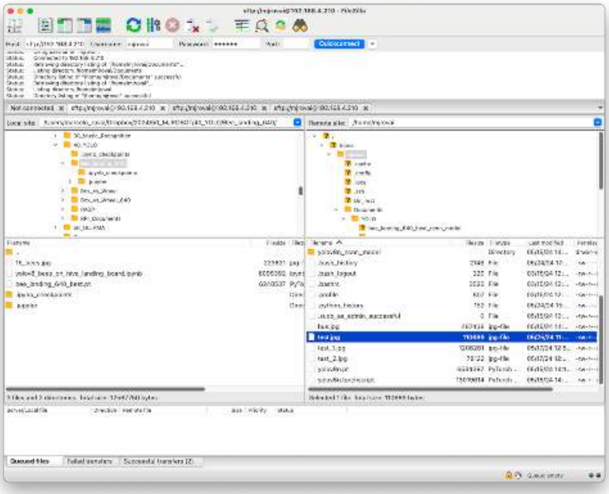
```
This is a test!
"Hello World!"
```

Transferring files using FTP

Transferring files using FTP, such as FileZilla FTP Client, is also possible. Follow the instructions, install the program for your Desktop OS, and use the Raspi IP address as the `Host`. For example:

```
sftp://192.168.4.210
```

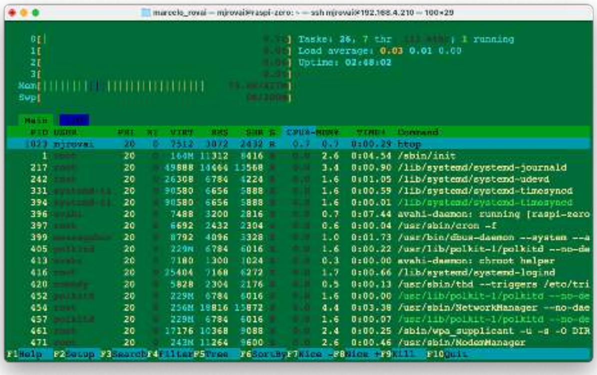
and enter your Raspi `username and password`. Pressing `Quickconnect` will open two windows, one for your host computer desktop (right) and another for the Raspi (left).



Increasing SWAP Memory

Using `htop`, a cross-platform interactive process viewer, you can easily monitor the resources running on your Raspi, such as the list of processes, the running CPUs, and the memory used in real-time. To lunch `hop`, enter with the command on the terminal:

```
htop
```



marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 100×29

```
    0[|                              [illegible]] Tasks: 26, 7 thr; [illegible]; 1 running
    1[                               [illegible]] Load average: 0.03 0.01 0.00
    2[                               [illegible]] Uptime: 02:48:02
    3[                               [illegible]]
  Mem[||||||||||||||||||||||||||     [illegible]]
  Swp[                               [illegible]]

  Main  [illegible]
    PID USER       PRI  NI  VIRT   RES   SHR S CPU%-MEM%   TIME+  Command
   1023 mjrovai     20   0  7512  3072  2432 R  0.7  0.7  0:00.29 htop
      1 root        20   0  164M 11312  8416 S  0.0  2.6  0:04.54 /sbin/init
    217 root        20   0 49888 14464 13568 S  0.0  3.4  0:00.90 /lib/systemd/systemd-journald
    242 root        20   0 26308  6784  4224 S  0.0  1.6  0:01.05 /lib/systemd/systemd-udevd
    331 systemd-ti  20   0 90580  6656  5888 S  0.0  1.6  0:00.59 /lib/systemd/systemd-timesyncd
    394 systemd-ti  20   0 90580  6656  5888 S  0.0  1.6  0:00.01 /lib/systemd/systemd-timesyncd
    396 avahi       20   0  7488  3200  2816 S  0.0  0.7  0:07.44 avahi-daemon: running [raspi-zero
    397 root        20   0  6692  2432  2304 S  0.0  0.6  0:00.04 /usr/sbin/cron -f
    399 messagebus  20   0  8792  4096  3328 S  0.0  1.0  0:01.73 /usr/bin/dbus-daemon --system --a
    405 polkitd     20   0  229M  6784  6016 S  0.0  1.6  0:00.22 /usr/lib/polkit-1/polkitd --no-de
    413 avahi       20   0  7180  1300  1024 S  0.0  0.3  0:00.00 avahi-daemon: chroot helper
    416 root        20   0 25404  7168  6272 S  0.0  1.7  0:00.66 /lib/systemd/systemd-logind
    420 nobody      20   0  5828  2304  2176 S  0.0  0.5  0:00.13 /usr/sbin/thd --triggers /etc/tri
    452 polkitd     20   0  229M  6784  6016 S  0.0  1.6  0:00.00 /usr/lib/polkit-1/polkitd --no-de
    454 root        20   0  256M 18816 15872 S  0.0  4.4  0:03.38 /usr/sbin/NetworkManager --no-dae
    457 polkitd     20   0  229M  6784  6016 S  0.0  1.6  0:00.07 /usr/lib/polkit-1/polkitd --no-de
    461 root        20   0 17176 10368  9088 S  0.0  2.4  0:00.25 /sbin/wpa_supplicant -u -s -O DIR
    471 root        20   0  243M 11264  9600 S  0.0  2.6  0:00.46 /usr/sbin/ModemManager
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Regarding memory, among the devices in the Raspberry Pi family, the Raspi-Zero has the smallest amount of SRAM (500 MB), compared to a selection of 2 GB to 8 GB on the Raspis 4 or 5. For any Raspi, it is possible to increase the memory available to the system with “Swap.” Swap memory, also known as swap space, is a technique used in computer operating systems to temporarily store data from RAM (Random Access Memory) on the SD card when the physical RAM is fully utilized. This allows the operating system (OS) to continue running even when RAM is full, which can prevent system crashes or slowdowns.

Swap memory benefits devices with limited RAM, such as the Raspi-Zero. Increasing swap can help run more demanding applications or processes, but it’s essential to balance this with the potential performance impact of frequent disk access.

By default, the Rapi-Zero’s SWAP (Swp) memory is only 100 MB, which is very small for running some more complex and demanding Machine Learning applications (for example, YOLO). Let’s increase it to 2 MB:

First, turn off swap-file:

```
sudo dphys-swapfile swapoff
```

Next, you should open and change the file `/etc/dphys-swapfile`. For that, we will use the nano:

```
sudo nano /etc/dphys-swapfile
```

Search for the **CONF_SWAPSIZE** variable (default is 200) and update it to **2000**:

```
CONF_SWAPSIZE=2000
```

And save the file.

Next, turn on the swapfile again and reboot the Raspi-zero:

```
sudo dphys-swapfile setup
sudo dphys-swapfile swapon
sudo reboot
```

When your device is rebooted (you should enter with the SSH again), you will realize that the maximum swap memory value shown on top is now something near 2 GB (in my case, 1.95 GB).

> To keep the *htop* running, you should open another terminal window to interact continuously with your Raspi.

Installing a Camera

The Raspi is an excellent device for computer vision applications; a camera is needed for it. We can install a standard USB webcam on the micro-USB port using a USB OTG adapter (Raspi-Zero and Raspi-5) or a camera module connected to the Raspi CSI (Camera Serial Interface) port.

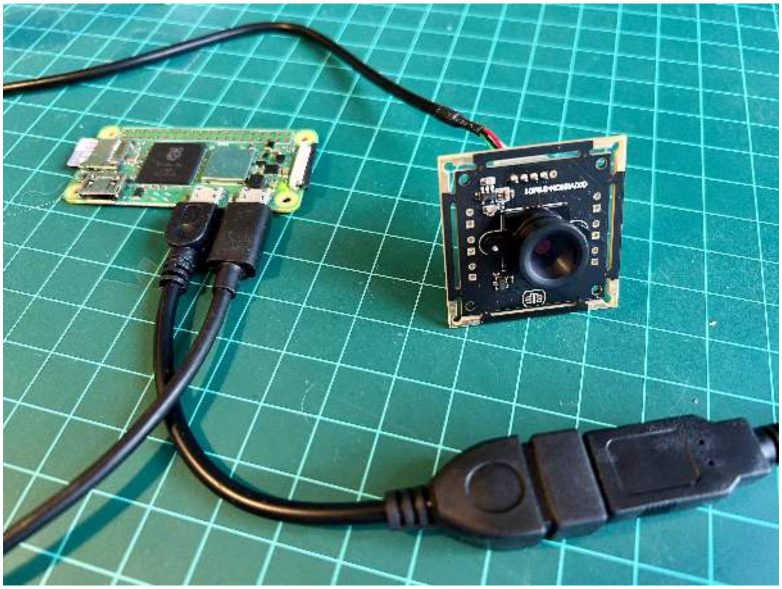
> USB Webcams generally have inferior quality to the camera modules that connect to the CSI port. They can also not be controlled using the `raspistill` and `raspivid` commands in the terminal or the `picamera` recording package in Python. Nevertheless, there may be reasons why you want to connect a USB camera to your Raspberry Pi, such as because of the benefit that it is much easier to set up multiple cameras with a single Raspberry Pi, long cables, or simply because you have such a camera on hand.

Installing a USB WebCam

1. Power off the Raspi:

```
sudo shutdown -h no
```

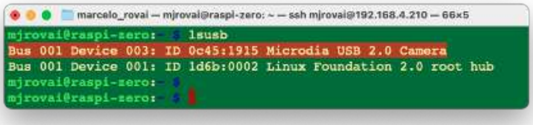
2. Connect the USB Webcam (USB Camera Module 30 fps, 1280 × 720) to your Raspi (In this example, I am using the Raspi-Zero, but the instructions work for all Raspis).



3. Power on again and run the SSH
4. To check if your USB camera is recognized, run:

```
lsusb
```

You should see your camera listed in the output.



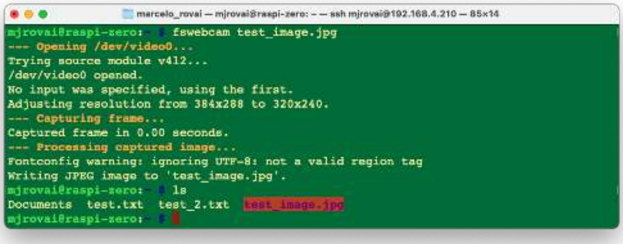
marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 66×5

```
mjrovai@raspi-zero:~ $ lsusb
Bus 001 Device 003: ID 0c45:1915 Microdia USB 2.0 Camera
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
mjrovai@raspi-zero:~ $
mjrovai@raspi-zero:~ $
```

5. To take a test picture with your USB camera, use:

```
fswebcam test_image.jpg
```

This will save an image named “test_image.jpg” in your current directory.



marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 85×14

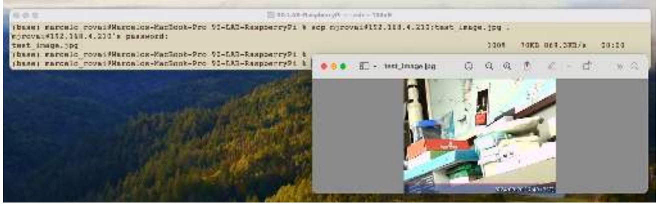
```
mjrovai@raspi-zero:~ $ fswebcam test_image.jpg
--- Opening /dev/video0...
Trying source module v4l2...
/dev/video0 opened.
No input was specified, using the first.
Adjusting resolution from 384x288 to 320x240.
--- Capturing frame...
Captured frame in 0.00 seconds.
--- Processing captured image...
Fontconfig warning: ignoring UTF-8: not a valid region tag
Writing JPEG image to 'test_image.jpg'.
mjrovai@raspi-zero:~ $ ls
Documents  test.txt  test_2.txt  test_image.jpg
mjrovai@raspi-zero:~ $
```

6. Since we are using SSH to connect to our Rapsi, we must transfer the image to our main computer so we can view it. We can use FileZilla or SCP for this:

Open a terminal **on your host computer** and run:

```
scp mjrovai@raspi-zero.local:~/test_image.jpg .
```

Replace "mjrovai" with your username and "raspi-zero" with Pi's hostname.

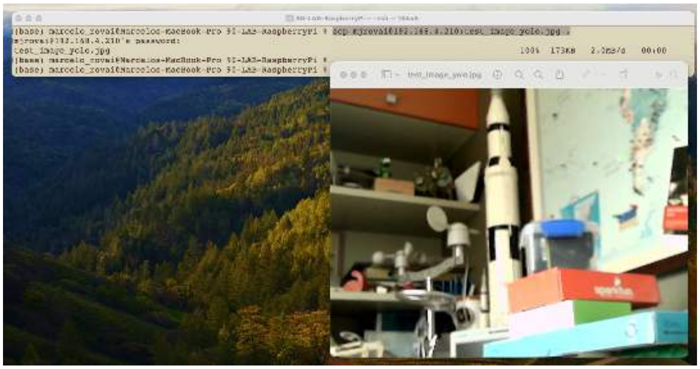


```
(base) marcelo_rovai@Marcelos-MacBook-Pro 92-LAB-RaspberryPi % scp mjrovai@192.168.4.210:test_image.jpg .
mjrovai@192.168.4.210's password:
test_image.jpg                                                          100%   10KB 869.3KB/s   00:00
(base) marcelo_rovai@Marcelos-MacBook-Pro 92-LAB-RaspberryPi %
(base) marcelo_rovai@Marcelos-MacBook-Pro 92-LAB-RaspberryPi %
```

7. If the image quality isn't satisfactory, you can adjust various settings; for example, define a resolution that is suitable for YOLO $(640x640)$:

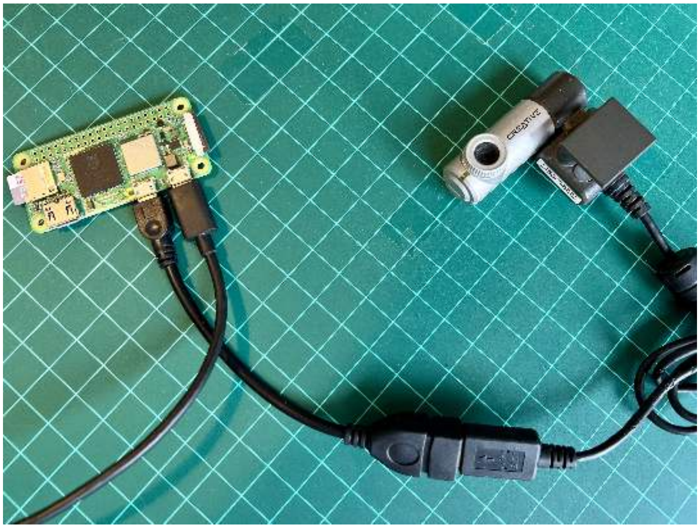
```
fswebcam -r 640x640 --no-banner test_image_yolo.jpg
```

This captures a higher-resolution image without the default banner.

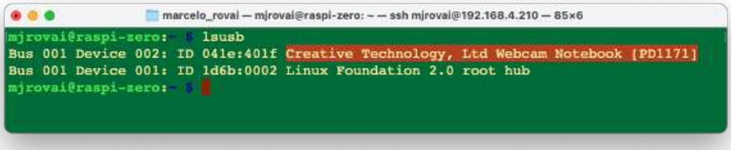


```
(base) marcelo_rovai@Marcelos-MacBook-Pro 92-LAB-RaspberryPi % scp mjrovai@192.168.4.210:test_image_yolo.jpg .
mjrovai@192.168.4.210's password:
test_image_yolo.jpg                                                     100%  173KB   2.0MB/s   00:00
(base) marcelo_rovai@Marcelos-MacBook-Pro 92-LAB-RaspberryPi %
(base) marcelo_rovai@Marcelos-MacBook-Pro 92-LAB-RaspberryPi %
```

An ordinary USB Webcam can also be used:



And verified using `lsusb`



marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 85×6

```
mjrovai@raspi-zero:~ $ lsusb
Bus 001 Device 002: ID 041e:401f Creative Technology, Ltd Webcam Notebook [PD1171]
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
mjrovai@raspi-zero:~ $
```

Video Streaming

For stream video (which is more resource-intensive), we can install and use mjpg-streamer:

First, install Git:

```
sudo apt install git
```

Now, we should install the necessary dependencies for mjpg-streamer, clone the repository, and proceed with the installation:

```
sudo apt install cmake libjpeg62-turbo-dev
git clone https://github.com/jacksonliam/mjpg-streamer.git
cd mjpg-streamer/mjpg-streamer-experimental
make
sudo make install
```

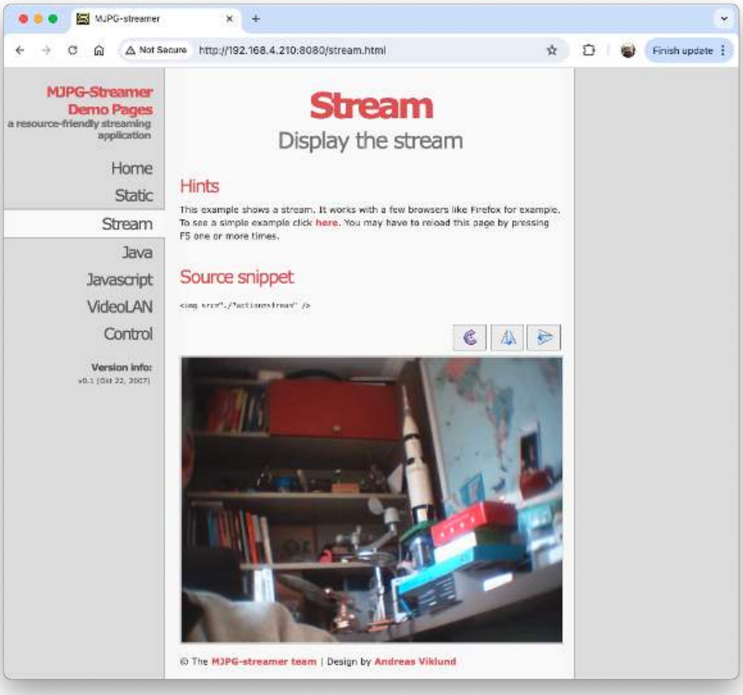
Then start the stream with:

```
mjpg_streamer -i "input_uvc.so" -o "output_http.so -w ./www"
```

We can then access the stream by opening a web browser and navigating to: `http://<your_pi_ip_address>:8080`. In my case: http://192.168.4.210:8080

We should see a webpage with options to view the stream. Click on the link that says "Stream" or try accessing:

```
http://<raspberry_pi_ip_address>:8080/?action=stream
```



This example shows a stream. It works with a few browsers like Firefox for example. To see a simple example click here. You may have to reload this page by pressing F5 one or more times.

```
<img src="./?action=stream" />
```

© The MJPG-streamer team | Design by Andreas Viklund

Installing a Camera Module on the CSI port

There are now several Raspberry Pi camera modules. The original 5-megapixel model was released in 2013, followed by an 8-megapixel Camera Module 2 that was later released in 2016. The latest camera model is the 12-megapixel Camera Module 3, released in 2023.

The original 5 MP camera (**Arducam OV5647**) is no longer available from Raspberry Pi but can be found from several alternative suppliers. Below is an example of such a camera on a Raspi-Zero.

Here is another example of a v2 Camera Module, which has a **Sony IMX219** 8-megapixel sensor:

Any camera module will work on the Raspberry Pis, but for that, the `configuration.txt` file must be updated:

```
sudo nano /boot/firmware/config.txt
```

At the bottom of the file, for example, to use the 5 MP Arducam OV5647 camera, add the line:

```
dtoverlay=ov5647,cam0
```

Or for the v2 module, which has the 8MP Sony IMX219 camera:

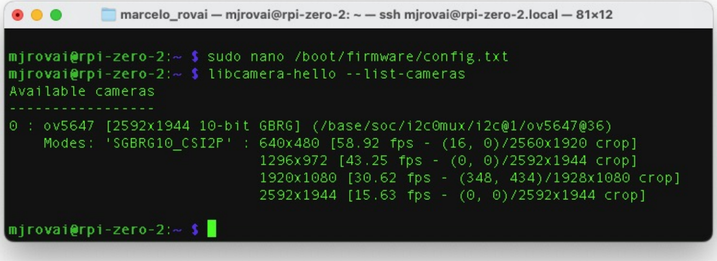
```
dtoverlay=imx219,cam0
```

Save the file (CTRL+O [ENTER] CRTL+X) and reboot the Raspi:

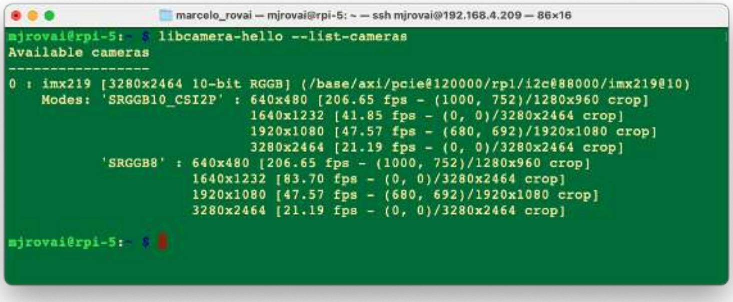
```
Sudo reboot
```

After the boot, you can see if the camera is listed:

```
libcamera-hello --list-cameras
```


marcelo_rovai — mjrovai@rpi-zero-2: ~ — ssh mjrovai@rpi-zero-2.local — 81×12

```
mjrovai@rpi-zero-2:~ $ sudo nano /boot/firmware/config.txt
mjrovai@rpi-zero-2:~ $ libcamera-hello --list-cameras
Available cameras
-----------------
0 : ov5647 [2592x1944 10-bit GBRG] (/base/soc/i2c0mux/i2c@1/ov5647@36)
    Modes: 'SGBRG10_CSI2P' : 640x480 [58.92 fps - (16, 0)/2560x1920 crop]
                             1296x972 [43.25 fps - (0, 0)/2592x1944 crop]
                             1920x1080 [30.62 fps - (348, 434)/1928x1080 crop]
                             2592x1944 [15.63 fps - (0, 0)/2592x1944 crop]

mjrovai@rpi-zero-2:~ $
```


marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@192.168.4.209 — 86×16

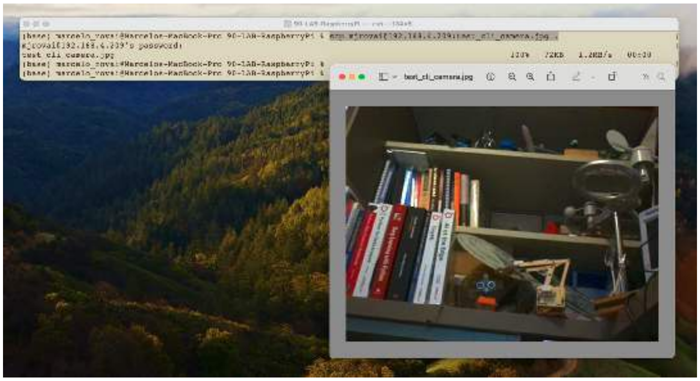
```
mjrovai@rpi-5:~ $ libcamera-hello --list-cameras
Available cameras
-----------------
0 : imx219 [3280x2464 10-bit RGGB] (/base/axi/pcie@120000/rp1/i2c@88000/imx219@10)
    Modes: 'SRGGB10_CSI2P' : 640x480 [206.65 fps - (1000, 752)/1280x960 crop]
                             1640x1232 [41.85 fps - (0, 0)/3280x2464 crop]
                             1920x1080 [47.57 fps - (680, 692)/1920x1080 crop]
                             3280x2464 [21.19 fps - (0, 0)/3280x2464 crop]
           'SRGGB8' : 640x480 [206.65 fps - (1000, 752)/1280x960 crop]
                      1640x1232 [83.70 fps - (0, 0)/3280x2464 crop]
                      1920x1080 [47.57 fps - (680, 692)/1920x1080 crop]
                      3280x2464 [21.19 fps - (0, 0)/3280x2464 crop]

mjrovai@rpi-5:~ $
```

> libcamera is an open-source software library that supports camera systems directly from the Linux operating system on Arm processors. It minimizes proprietary code running on the Broadcom GPU.

Let's capture a jpeg image with a resolution of 640 × 480 for testing and save it to a file named `test_cli_camera.jpg`

```
rpicam-jpeg --output test_cli_camera.jpg --width 640 --height 480
```

if we want to see the file saved, we should use `ls -f`, which lists all current directory content in long format. As before, we can use scp to view the image:



```
(base) marcelo_rovai@Marcelos-MacBook-Pro 90-IAB-RaspberryPi % scp mjrovai@192.168.4.209:test_cli_camera.jpg .
mjrovai@192.168.4.209's password:
test_cli_camera.jpg                                             100%   72KB   1.2MB/s   00:00
(base) marcelo_rovai@Marcelos-MacBook-Pro 90-IAB-RaspberryPi %
(base) marcelo_rovai@Marcelos-MacBook-Pro 90-IAB-RaspberryPi %
```

Running the Raspi Desktop remotely

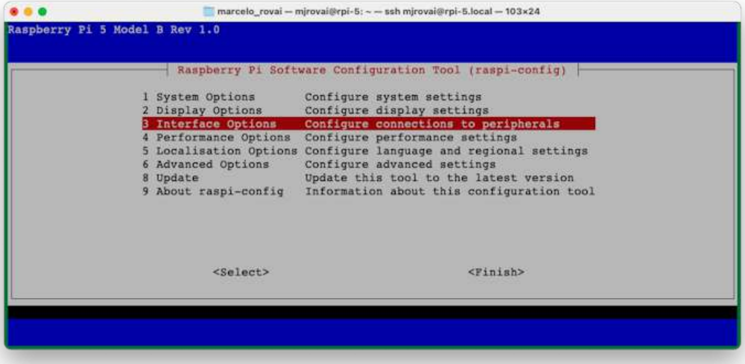
While we've primarily interacted with the Raspberry Pi using terminal commands via SSH, we can access the whole graphical desktop environment remotely if we have installed the complete Raspberry Pi OS (for example, `Raspberry Pi OS (64-bit)`. This can be particularly useful for tasks that benefit from a visual interface. To enable this functionality, we must set up a VNC (Virtual Network Computing) server on the Raspberry Pi. Here's how to do it:

1. Enable the VNC Server:

   - Connect to your Raspberry Pi via SSH.
   - Run the Raspberry Pi configuration tool by entering:

```
sudo raspi-config
```

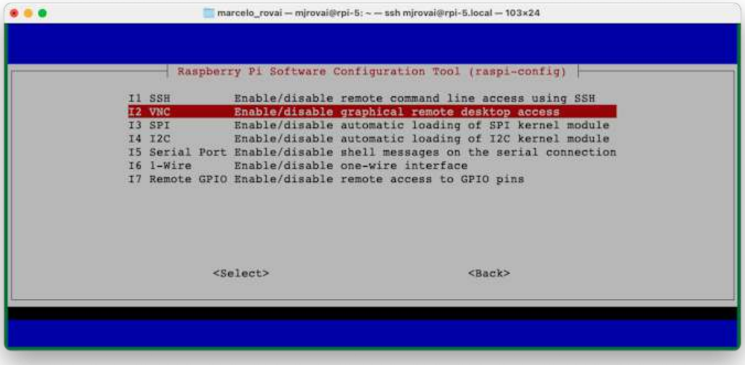
   - Navigate to `Interface Options` using the arrow keys.



marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 103×24

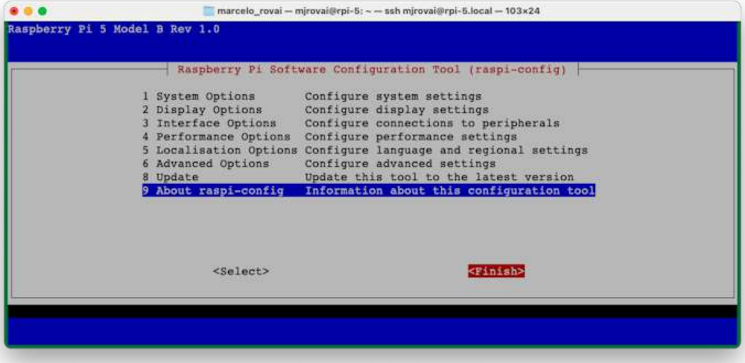
```
Raspberry Pi 5 Model B Rev 1.0

         Raspberry Pi Software Configuration Tool (raspi-config)

    1 System Options       Configure system settings
    2 Display Options      Configure display settings
    3 Interface Options    Configure connections to peripherals
    4 Performance Options  Configure performance settings
    5 Localisation Options Configure language and regional settings
    6 Advanced Options     Configure advanced settings
    8 Update               Update this tool to the latest version
    9 About raspi-config   Information about this configuration tool

              <Select>                          <Finish>
```

- Select `VNC` and `Yes` to enable the VNC server.



```
marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 103×24

Raspberry Pi Software Configuration Tool (raspi-config)

I1 SSH         Enable/disable remote command line access using SSH
I2 VNC         Enable/disable graphical remote desktop access
I3 SPI         Enable/disable automatic loading of SPI kernel module
I4 I2C         Enable/disable automatic loading of I2C kernel module
I5 Serial Port Enable/disable shell messages on the serial connection
I6 1-Wire      Enable/disable one-wire interface
I7 Remote GPIO Enable/disable remote access to GPIO pins

            <Select>                          <Back>
```

- Exit the configuration tool, saving changes when prompted.



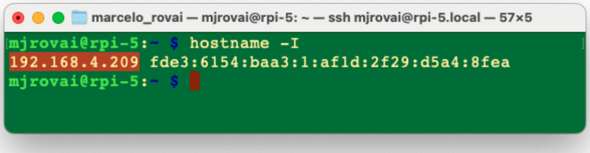
```
marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 103×24
Raspberry Pi 5 Model B Rev 1.0

Raspberry Pi Software Configuration Tool (raspi-config)

1 System Options       Configure system settings
2 Display Options      Configure display settings
3 Interface Options    Configure connections to peripherals
4 Performance Options  Configure performance settings
5 Localisation Options Configure language and regional settings
6 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool

            <Select>                          <Finish>
```

2. Install a VNC Viewer on Your Computer:

   - Download and install a VNC viewer application on your main computer. Popular options include RealVNC Viewer, TightVNC, or VNC Viewer by RealVNC. We will install VNC Viewer by RealVNC.

3. Once installed, you should confirm the Raspi IP address. For example, on the terminal, you can use:

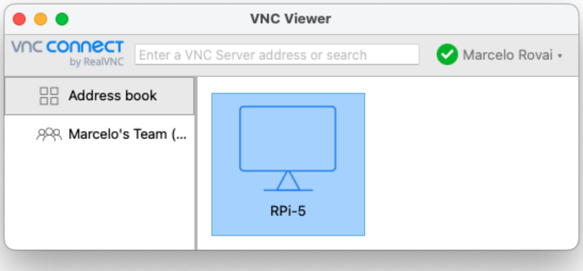
```
hostname -I
```



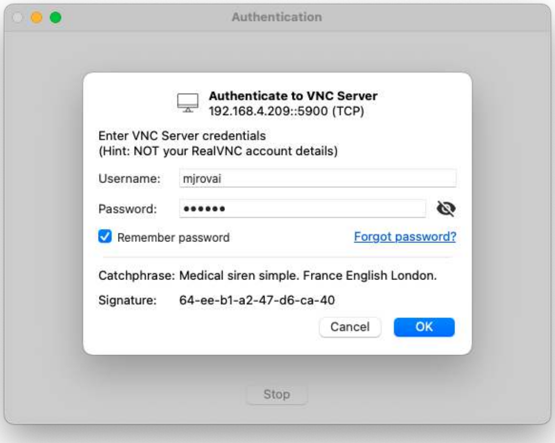
marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 57×5

```
mjrovai@rpi-5:~ $ hostname -I
192.168.4.209 fde3:6154:baa3:1:af1d:2f29:d5a4:8fea
mjrovai@rpi-5:~ $
```

4. Connect to Your Raspberry Pi:
   - Open your VNC viewer application.



- Enter your Raspberry Pi's IP address and hostname.
- When prompted, enter your Raspberry Pi's username and password.



Authentication

Authenticate to VNC Server
192.168.4.209::5900 (TCP)

Enter VNC Server credentials
(Hint: NOT your RealVNC account details)

Username: mjrovai

Password: ••••••

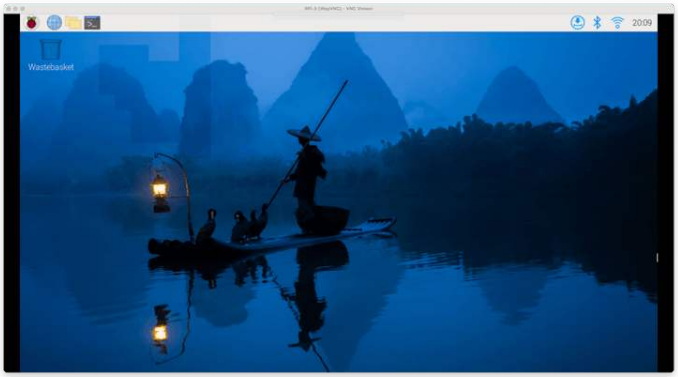
☑ Remember password

Forgot password?

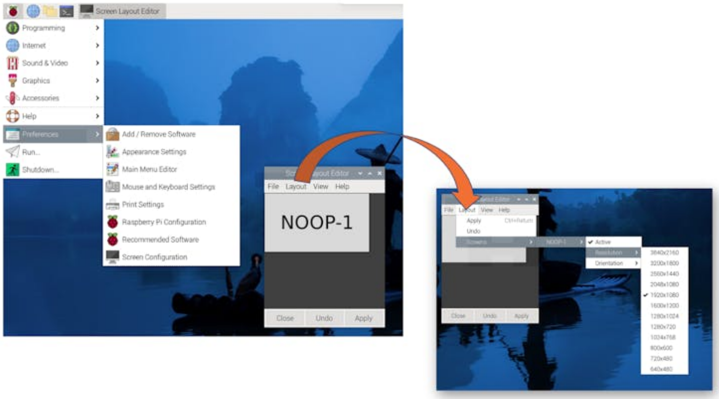
Catchphrase: Medical siren simple. France English London.

Signature: 64-ee-b1-a2-47-d6-ca-40

5. The Raspberry Pi 5 Desktop should appear on your computer monitor.



6. Adjust Display Settings (if needed):
   - Once connected, adjust the display resolution for optimal viewing. This can be done through the Raspberry Pi's desktop settings or by modifying the config.txt file.
   - Let's do it using the desktop settings. Reach the menu (the Raspberry Icon at the left upper corner) and select the best screen definition for your monitor:



Updating and Installing Software

1. Update your system:

```
sudo apt update && sudo apt upgrade -y
```

2. Install essential software:

```
sudo apt install python3-pip -y
```

3. Enable pip for Python projects:

```
sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED
```

Model-Specific Considerations

Raspberry Pi Zero (Raspi-Zero)

- Limited processing power, best for lightweight projects
- It is better to use a headless setup (SSH) to conserve resources.
- Consider increasing swap space for memory-intensive tasks.
- It can be used for Image Classification and Object Detection Labs but not for the LLM (SLM).

Raspberry Pi 4 or 5 (Raspi-4 or Raspi-5)

- Suitable for more demanding projects, including AI and machine learning.
- It can run the whole desktop environment smoothly.
- Raspi-4 can be used for Image Classification and Object Detection Labs but will not work well with LLMs (SLM).
- For Raspi-5, consider using an active cooler for temperature management during intensive tasks, as in the LLMs (SLMs) lab.

Remember to adjust your project requirements based on the specific Raspberry Pi model you're using. The Raspi-Zero is great for low-power, space-constrained projects, while the Raspi-4 or 5 models are better suited for more computationally intensive tasks.

Image Classification

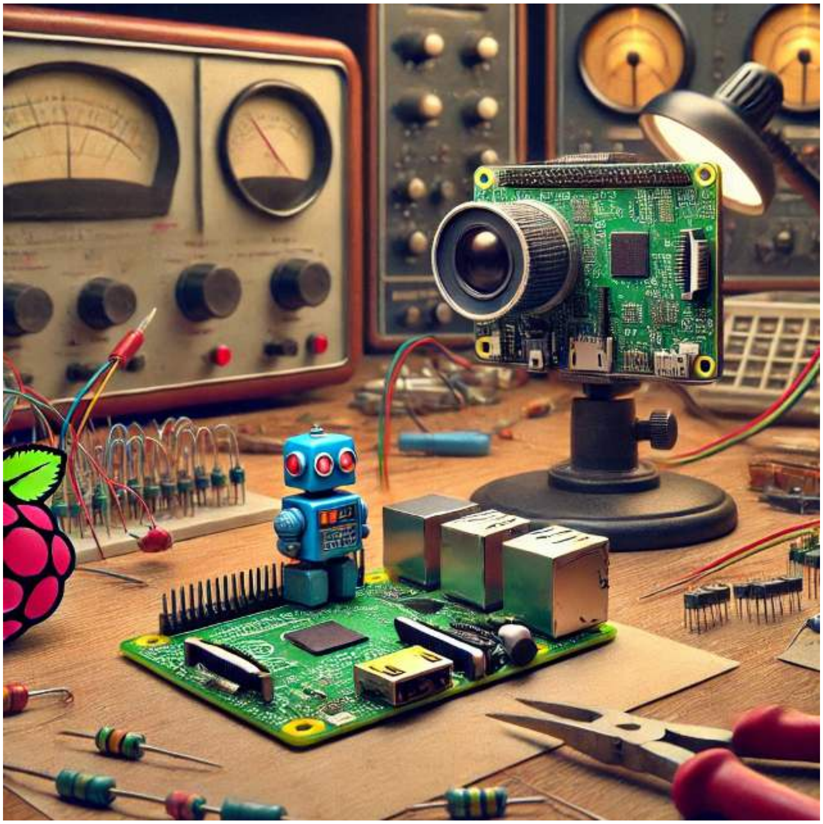


Figure 20.18: *DALL·E prompt - A cover image for an 'Image Classification' chapter in a Raspberry Pi tutorial, designed in the same vintage 1950s electronics lab style as previous covers. The scene should feature a Raspberry Pi connected to a camera module, with the camera capturing a photo of the small blue robot provided by the user. The robot should be placed on a workbench, surrounded by classic lab tools like soldering irons, resistors, and wires. The lab background should include vintage equipment like oscilloscopes and tube radios, maintaining the detailed and nostalgic feel of the era. No text or logos should be included.*

Overview

Image classification is a fundamental task in computer vision that involves categorizing an image into one of several predefined classes. It's a cornerstone of artificial intelligence, enabling machines to interpret and understand visual information in a way that mimics human perception.

Image classification refers to assigning a label or category to an entire image based on its visual content. This task is crucial in computer vision and has numerous applications across various industries. Image classification's importance lies in its ability to automate visual understanding tasks that would otherwise require human intervention.

Applications in Real-World Scenarios

Image classification has found its way into numerous real-world applications, revolutionizing various sectors:

- Healthcare: Assisting in medical image analysis, such as identifying abnormalities in X-rays or MRIs.
- Agriculture: Monitoring crop health and detecting plant diseases through aerial imagery.
- Automotive: Enabling advanced driver assistance systems and autonomous vehicles to recognize road signs, pedestrians, and other vehicles.
- Retail: Powering visual search capabilities and automated inventory management systems.
- Security and Surveillance: Enhancing threat detection and facial recognition systems.
- Environmental Monitoring: Analyzing satellite imagery for deforestation, urban planning, and climate change studies.

Advantages of Running Classification on Edge Devices like Raspberry Pi

Implementing image classification on edge devices such as the Raspberry Pi offers several compelling advantages:

1. Low Latency: Processing images locally eliminates the need to send data to cloud servers, significantly reducing response times.
2. Offline Functionality: Classification can be performed without an internet connection, making it suitable for remote or connectivity-challenged environments.
3. Privacy and Security: Sensitive image data remains on the local device, addressing data privacy concerns and compliance requirements.
4. Cost-Effectiveness: Eliminates the need for expensive cloud computing resources, especially for continuous or high-volume classification tasks.
5. Scalability: Enables distributed computing architectures where multiple devices can work independently or in a network.
6. Energy Efficiency: Optimized models on dedicated hardware can be more energy-efficient than cloud-based solutions, which is crucial for battery-powered or remote applications.
7. Customization: Deploying specialized or frequently updated models tailored to specific use cases is more manageable.

We can create more responsive, secure, and efficient computer vision solutions by leveraging the power of edge devices like Raspberry Pi for image classification. This approach opens up new possibilities for integrating intelligent visual processing into various applications and environments.

In the following sections, we'll explore how to implement and optimize image classification on the Raspberry Pi, harnessing these advantages to create powerful and efficient computer vision systems.

Setting Up the Environment

Updating the Raspberry Pi

First, ensure your Raspberry Pi is up to date:

```
sudo apt update
sudo apt upgrade -y
```

Installing Required Libraries

Install the necessary libraries for image processing and machine learning:

```
sudo apt install python3-pip
sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED
pip3 install --upgrade pip
```

Setting up a Virtual Environment (Optional but Recommended)

Create a virtual environment to manage dependencies:

```
python3 -m venv ~/tflite
source ~/tflite/bin/activate
```

Installing TensorFlow Lite

We are interested in performing **inference**, which refers to executing a TensorFlow Lite model on a device to make predictions based on input data. To perform an inference with a TensorFlow Lite model, we must run it through an **interpreter**. The TensorFlow Lite interpreter is designed to be lean and fast. The interpreter uses a static graph ordering and a custom (less-dynamic) memory allocator to ensure minimal load, initialization, and execution latency.

We'll use the TensorFlow Lite runtime for Raspberry Pi, a simplified library for running machine learning models on mobile and embedded devices, without including all TensorFlow packages.

```
pip install tflite_runtime --no-deps
```

> The wheel installed: `tflite_runtime-2.14.0-cp311-cp311-manylinux_-2_34_aarch64.whl`

Installing Additional Python Libraries

Install required Python libraries for use with Image Classification:

If you have another version of Numpy installed, first uninstall it.

```
pip3 uninstall numpy
```

Install `version 1.23.2`, which is compatible with the tflite_runtime.

```
 pip3 install numpy==1.23.2
```

```
pip3 install Pillow matplotlib
```

Creating a working directory:

If you are working on the Raspi-Zero with the minimum OS (No Desktop), you may not have a user-pre-defined directory tree (you can check it with `ls`. So, let's create one:

```
mkdir Documents
cd Documents/
mkdir TFLITE
cd TFLITE/
mkdir IMG_CLASS
cd IMG_CLASS
mkdir models
cd models
```

On the Raspi-5, the /Documents should be there.

Get a pre-trained Image Classification model:

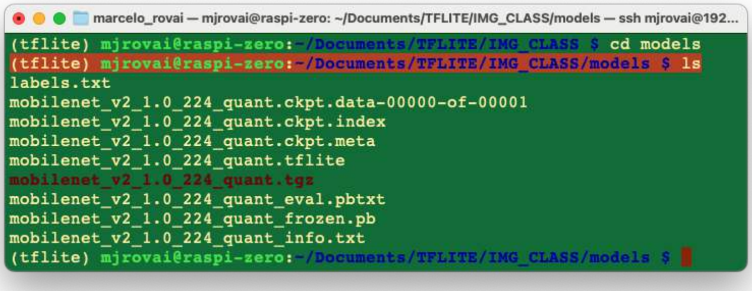
An appropriate pre-trained model is crucial for successful image classification on resource-constrained devices like the Raspberry Pi. **MobileNet** is designed for mobile and embedded vision applications with a good balance between accuracy and speed. Versions: MobileNetV1, MobileNetV2, MobileNetV3. Let's download the V2:

```
# One long line, split with backslash \
wget https://storage.googleapis.com/download.tensorflow.org/\
models/tflite_11_05_08/mobilenet_v2_1.0_224_quant.tgz

tar xzf mobilenet_v2_1.0_224_quant.tgz
```

Get its labels:

```
# One long line, split with backslash \
wget https://github.com/Mjrovai/EdgeML-with-Raspberry-Pi/blob/\
main/IMG_CLASS/models/labels.txt
```

In the end, you should have the models in its directory:


marcelo_rovai — mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS/models — ssh mjrovai@192...

```
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ cd models
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS/models $ ls
labels.txt
mobilenet_v2_1.0_224_quant.ckpt.data-00000-of-00001
mobilenet_v2_1.0_224_quant.ckpt.index
mobilenet_v2_1.0_224_quant.ckpt.meta
mobilenet_v2_1.0_224_quant.tflite
mobilenet_v2_1.0_224_quant.tgz
mobilenet_v2_1.0_224_quant_eval.pbtxt
mobilenet_v2_1.0_224_quant_frozen.pb
mobilenet_v2_1.0_224_quant_info.txt
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS/models $
```

We will only need the `mobilenet_v2_1.0_224_quant.tflite` model and the `labels.txt`. You can delete the other files.

Setting up Jupyter Notebook (Optional)

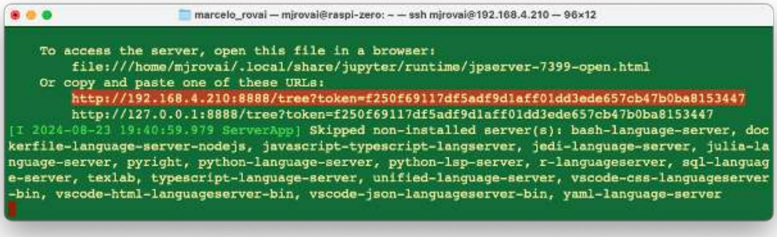
If you prefer using Jupyter Notebook for development:

```
pip3 install jupyter
jupyter notebook --generate-config
```

To run Jupyter Notebook, run the command (change the IP address for yours):

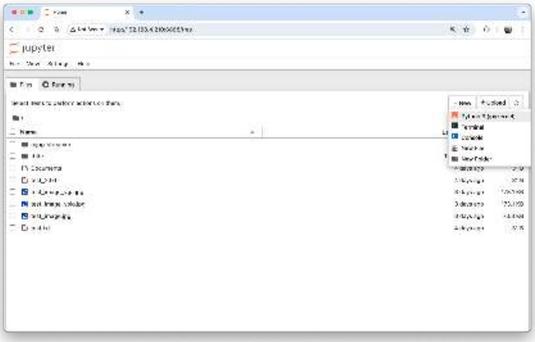
```
jupyter notebook --ip=192.168.4.210 --no-browser
```

On the terminal, you can see the local URL address to open the notebook:


marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 96×12

```
    To access the server, open this file in a browser:
        file:///home/mjrovai/.local/share/jupyter/runtime/jpserver-7399-open.html
    Or copy and paste one of these URLs:
        http://192.168.4.210:8888/tree?token=f250f69117df5adf9d1aff01dd3ede657cb47b0ba8153447
        http://127.0.0.1:8888/tree?token=f250f69117df5adf9d1aff01dd3ede657cb47b0ba8153447
[I 2024-08-23 19:40:59.979 ServerApp] Skipped non-installed server(s): bash-language-server, doc
kerfile-language-server-nodejs, javascript-typescript-langserver, jedi-language-server, julia-la
nguage-server, pyright, python-language-server, python-lsp-server, r-languageserver, sql-languag
e-server, texlab, typescript-language-server, unified-language-server, vscode-css-languageserver
-bin, vscode-html-languageserver-bin, vscode-json-languageserver-bin, yaml-language-server
```

You can access it from another device by entering the Raspberry Pi's IP address and the provided token in a web browser (you can copy the token from the terminal).



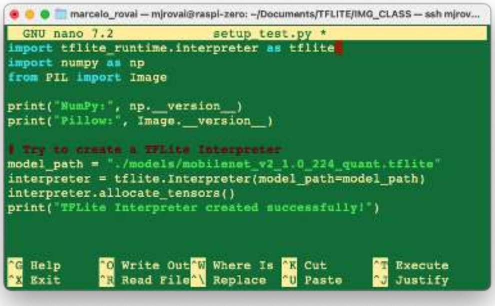
Define your working directory in the Raspi and create a new Python 3 notebook.

Verifying the Setup

Test your setup by running a simple Python script:

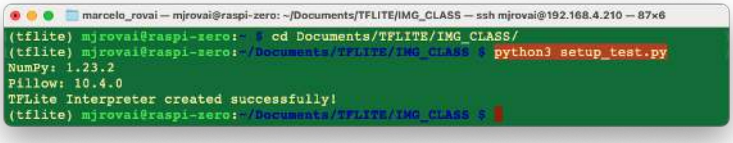
```
import tflite_runtime.interpreter as tflite
import numpy as np
from PIL import Image

print("NumPy:", np.__version__)
print("Pillow:", Image.__version__)

# Try to create a TFLite Interpreter
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
print("TFLite Interpreter created successfully!")
```

You can create the Python script using nano on the terminal, saving it with CTRL+0 + ENTER + CTRL+X



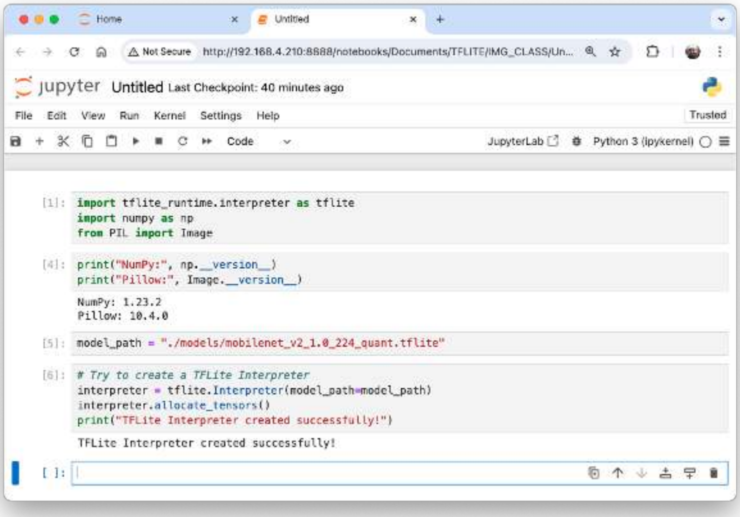
```
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS — ssh mjrov...
  GNU nano 7.2                 setup_test.py *
import tflite_runtime.interpreter as tflite
import numpy as np
from PIL import Image

print("NumPy:", np.__version__)
print("Pillow:", Image.__version__)

# Try to create a TFLite Interpreter
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
print("TFLite Interpreter created successfully!")



^G Help       ^O Write Out^W Where Is ^K Cut        ^T Execute
^X Exit       ^R Read File^\ Replace  ^U Paste      ^J Justify
```

And run it with the command:


marcelo_rovai — mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS — ssh mjrovai@192.168.4.210 — 87×6

```
(tflite) mjrovai@raspi-zero:~ $ cd Documents/TFLITE/IMG_CLASS/
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ python3 setup_test.py
NumPy: 1.23.2
Pillow: 10.4.0
TFLite Interpreter created successfully!
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $
```

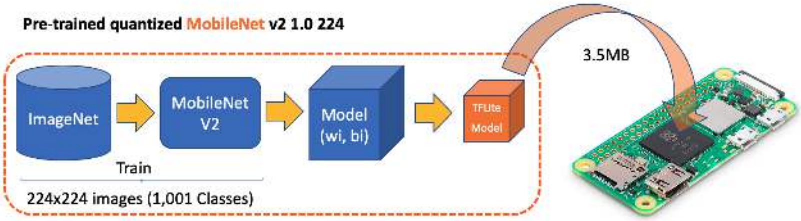
Or you can run it directly on the Notebook:


Home
Untitled
Not Secure http://192.168.4.210:8888/notebooks/Documents/TFLITE/IMG_CLASS/Un...
jupyter Untitled Last Checkpoint: 40 minutes ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
[1]: import tflite_runtime.interpreter as tflite
     import numpy as np
     from PIL import Image

[4]: print("NumPy:", np.__version__)
     print("Pillow:", Image.__version__)

     NumPy: 1.23.2
     Pillow: 10.4.0

[5]: model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"

[6]: # Try to create a TFLite Interpreter
     interpreter = tflite.Interpreter(model_path=model_path)
     interpreter.allocate_tensors()
     print("TFLite Interpreter created successfully!")

     TFLite Interpreter created successfully!

[ ]:
```

Making inferences with Mobilenet V2

In the last section, we set up the environment, including downloading a popular pre-trained model, Mobilenet V2, trained on ImageNet's 224 × 224 images (1.2 million) for 1,001 classes (1,000 object categories plus 1 background). The model was converted to a compact 3.5 MB TensorFlow Lite format, making it suitable for the limited storage and memory of a Raspberry Pi.


Pre-trained quantized MobileNet v2 1.0 224
ImageNet
MobileNet V2
Model (wi, bi)
TFLite Model
Train
224x224 images (1,001 Classes)
3.5MB

Let's start a new notebook to follow all the steps to classify one image:

Import the needed libraries:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tflite_runtime.interpreter as tflite
```

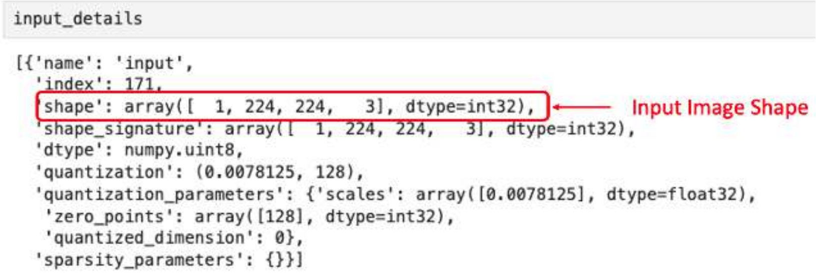
Load the TFLite model and allocate tensors:

```
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
```

Get input and output tensors.

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

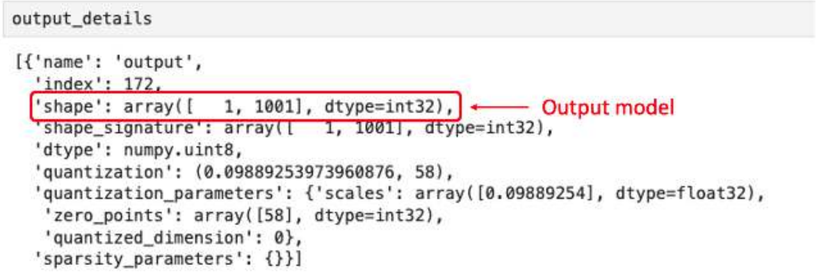
Input details will give us information about how the model should be fed with an image. The shape of (1, 224, 224, 3) informs us that an image with dimensions $(224 \times 224 \times 3)$ should be input one by one (Batch Dimension: 1).



```
input_details
```

```
[{'name': 'input',
  'index': 171,
  'shape': array([  1, 224, 224,   3], dtype=int32),
  'shape_signature': array([  1, 224, 224,   3], dtype=int32),
  'dtype': numpy.uint8,
  'quantization': (0.0078125, 128),
  'quantization_parameters': {'scales': array([0.0078125], dtype=float32),
   'zero_points': array([128], dtype=int32),
   'quantized_dimension': 0},
  'sparsity_parameters': {}}]
```

The **output details** show that the inference will result in an array of 1,001 integer values. Those values result from the image classification, where each value is the probability of that specific label being related to the image.



```
output_details
```

```
[{'name': 'output',
  'index': 172,
  'shape': array([   1, 1001], dtype=int32),
  'shape_signature': array([   1, 1001], dtype=int32),
  'dtype': numpy.uint8,
  'quantization': (0.09889253973960876, 58),
  'quantization_parameters': {'scales': array([0.09889254], dtype=float32),
   'zero_points': array([58], dtype=int32),
   'quantized_dimension': 0},
  'sparsity_parameters': {}}]
```

Let’s also inspect the dtype of input details of the model

```
input_dtype = input_details[0]['dtype']
input_dtype
```

```
dtype('uint8')
```

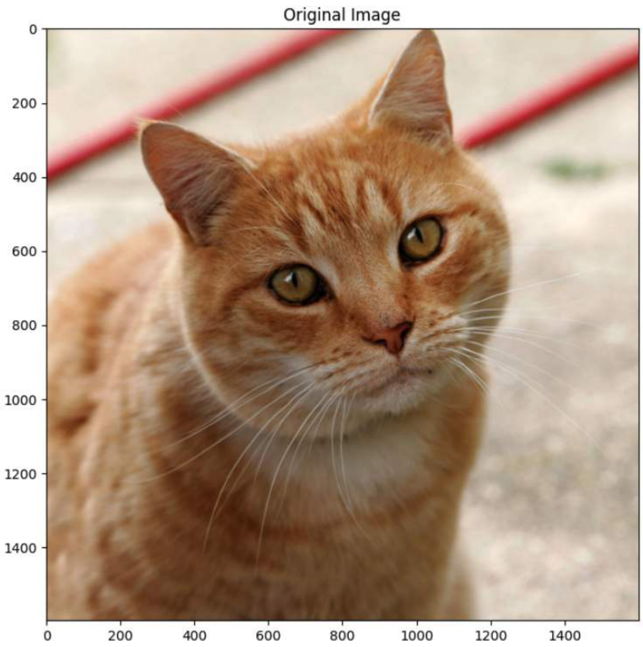
This shows that the input image should be raw pixels (0 - 255).

Let's get a test image. You can transfer it from your computer or download one for testing. Let's first create a folder under our working directory:

```
mkdir images
cd images
wget https://upload.wikimedia.org/wikipedia/commons/3/3a/Cat03.jpg
```

Let's load and display the image:

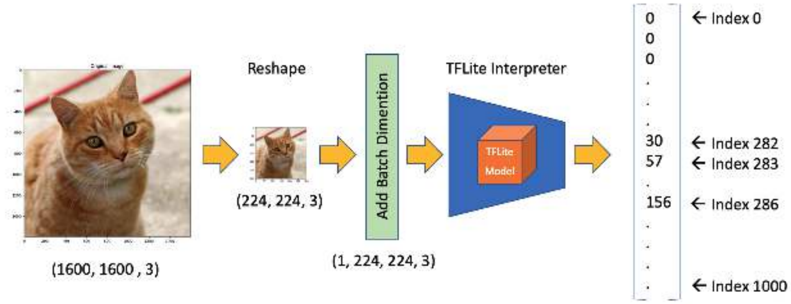
```
# Load he image
img_path = "./images/Cat03.jpg"
img = Image.open(img_path)

# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(img)
plt.title("Original Image")
plt.show()
```



We can see the image size running the command:

```
width, height = img.size
```

That shows us that the image is an RGB image with a width of 1600 and a height of 1600 pixels. So, to use our model, we should reshape it to (224, 224, 3) and add a batch dimension of 1, as defined in input details: (1, 224, 224, 3). The inference result, as shown in output details, will be an array with a 1001 size, as shown below:



So, let's reshape the image, add the batch dimension, and see the result:

```
img = img.resize((input_details[0]['shape'][1],
                  input_details[0]['shape'][2]))
input_data = np.expand_dims(img, axis=0)
input_data.shape
```

The input_data shape is as expected: (1, 224, 224, 3)

Let's confirm the dtype of the input data:

```
input_data.dtype
```

```
dtype('uint8')
```

The input data dtype is 'uint8', which is compatible with the dtype expected for the model.

Using the input_data, let's run the interpreter and get the predictions (output):

```
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
predictions = interpreter.get_tensor(output_details[0]
                                     ['index'])[0]
```

The prediction is an array with 1001 elements. Let's get the Top-5 indices where their elements have high values:

```
top_k_results = 5
top_k_indices = np.argsort(predictions)[::-1][:top_k_results]
top_k_indices
```

The top_k_indices is an array with 5 elements: `array([283, 286, 282])`

So, 283, 286, 282, 288, and 479 are the image's most probable classes. Having the index, we must find to what class it appoints (such as car, cat, or dog). The text file downloaded with the model has a label associated with each index from 0 to 1,000. Let's use a function to load the .txt file as a list:

```
def load_labels(filename):
    with open(filename, 'r') as f:
        return [line.strip() for line in f.readlines()]
```

And get the list, printing the labels associated with the indexes:

```
labels_path = "./models/labels.txt"
labels = load_labels(labels_path)

print(labels[286])
print(labels[283])
print(labels[282])
print(labels[288])
print(labels[479])
```

As a result, we have:

```
Egyptian cat
tiger cat
tabby
lynx
carton
```

At least the four top indices are related to felines. The **prediction** content is the probability associated with each one of the labels. As we saw on output details, those values are quantized and should be dequantized and apply softmax.

```
scale, zero_point = output_details[0]['quantization']
dequantized_output = (predictions.astype(np.float32) -
                      zero_point) * scale
exp_output = np.exp(dequantized_output -
             np.max(dequantized_output))
probabilities = exp_output / np.sum(exp_output)
```

Let's print the top-5 probabilities:

```
print (probabilities[286])
print (probabilities[283])
print (probabilities[282])
print (probabilities[288])
print (probabilities[479])
```

```
0.27741462
0.3732285
0.16919471
0.10319158
0.023410844
```

For clarity, let's create a function to relate the labels with the probabilities:

```
for i in range(top_k_results):
    print("\t{:20}: {}%".format(
        labels[top_k_indices[i]],
        (int(probabilities[top_k_indices[i]]*100))))
```

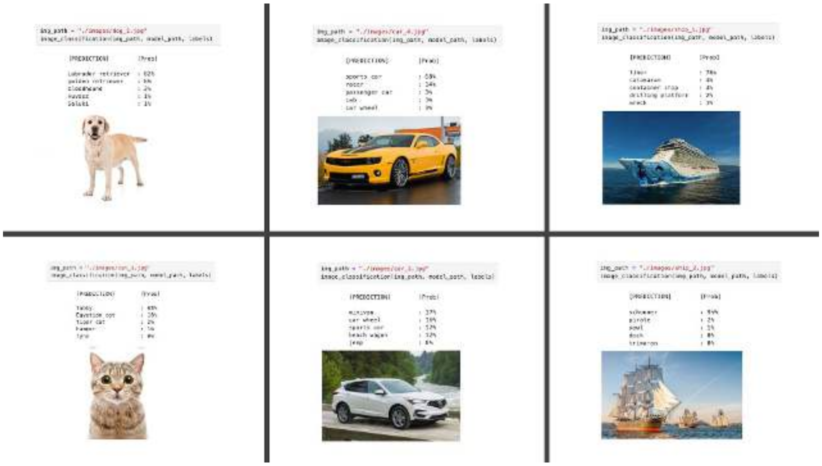
```
tiger cat           : 37%
Egyptian cat        : 27%
tabby               : 16%
lynx                : 10%
carton              : 2%
```

Define a general Image Classification function

Let's create a general function to give an image as input, and we get the Top-5 possible classes:

```
def image_classification(img_path, model_path, labels,
                         top_k_results=5):
    # load the image
    img = Image.open(img_path)
    plt.figure(figsize=(4, 4))
    plt.imshow(img)
    plt.axis('off')

    # Load the TFLite model
    interpreter = tflite.Interpreter(model_path=model_path)
    interpreter.allocate_tensors()

    # Get input and output tensors
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    # Preprocess
    img = img.resize((input_details[0]['shape'][1],
                      input_details[0]['shape'][2]))
    input_data = np.expand_dims(img, axis=0)

    # Inference on Raspi-Zero
    interpreter.set_tensor(input_details[0]['index'], input_data)
    interpreter.invoke()

    # Obtain results and map them to the classes
    predictions = interpreter.get_tensor(output_details[0]
                                         ['index'])[0]

    # Get indices of the top k results
    top_k_indices = np.argsort(predictions)[::-1][:top_k_results]

    # Get quantization parameters
    scale, zero_point = output_details[0]['quantization']

    # Dequantize the output and apply softmax
    dequantized_output = (predictions.astype(np.float32) -
                          zero_point) * scale
    exp_output = np.exp(dequantized_output -
                 np.max(dequantized_output))
    probabilities = exp_output / np.sum(exp_output)

    print("\n\t[PREDICTION]        [Prob]\n")
    for i in range(top_k_results):
        print("\t{:20}: {}%".format(
            labels[top_k_indices[i]],
            (int(probabilities[top_k_indices[i]]*100))))
```

And loading some images for testing, we have:



```
img_path = "./images/dog_1.jpg"
image_classification(img_path, model_path, labels)
```

```
[PREDICTION]          [Prob]

Labrador retriever    : 82%
golden retriever      : 6%
bloodhound            : 2%
kuvasz                : 1%
Saluki                : 1%
```

```
img_path = "./images/car_4.jpg"
image_classification(img_path, model_path, labels)
```

```
[PREDICTION]          [Prob]

sports car            : 58%
racer                 : 34%
passenger car         : 3%
cab                   : 3%
car wheel             : 0%
```

```
img_path = "./images/ship_1.jpg"
image_classification(img_path, model_path, labels)
```

```
[PREDICTION]          [Prob]

liner                 : 78%
catamaran             : 4%
container ship        : 4%
drilling platform     : 2%
wreck                 : 3%
```

```
img_path = "./images/cat_1.jpg"
image_classification(img_path, model_path, labels)
```

```
[PREDICTION]          [Prob]

tabby                 : 45%
Egyptian cat          : 35%
tiger cat             : 2%
hamper                : 1%
lynx                  : 0%
```

```
img_path = "./images/car_1.jpg"
image_classification(img_path, model_path, labels)
```

```
[PREDICTION]          [Prob]

minivan               : 37%
car wheel             : 18%
sports car            : 12%
beach wagon           : 12%
jeep                  : 6%
```

```
img_path = "./images/ship_2.jpg"
image_classification(img_path, model_path, labels)
```

```
[PREDICTION]          [Prob]

schooner              : 95%
pirate                : 2%
yawl                  : 1%
dock                  : 0%
trimaran              : 0%
```

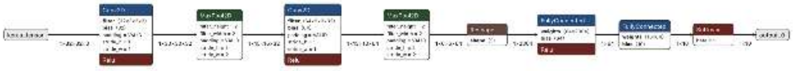
Testing with a model trained from scratch

Let's get a TFLite model trained from scratch. For that, you can follow the Notebook:

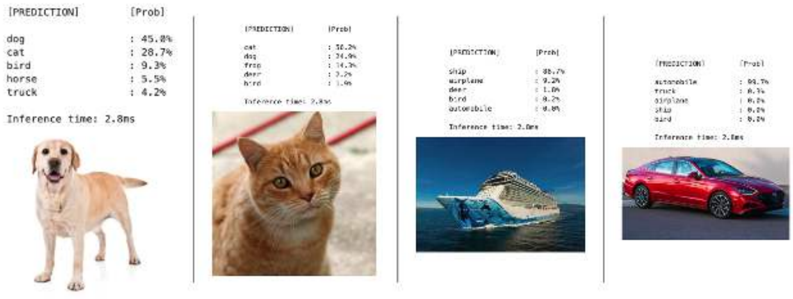
CNN to classify Cifar-10 dataset

In the notebook, we trained a model using the CIFAR10 dataset, which contains 60,000 images from 10 classes of CIFAR (*airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck*). CIFAR has 32 × 32 color images (3 color channels) where the objects are not centered and can have the object with a background, such as airplanes that might have a cloudy sky behind them! In short, small but real images.

The CNN trained model (*cifar10_model.keras*) had a size of 2.0MB. Using the *TFLite Converter*, the model *cifar10.tflite* became with 674MB (around 1/3 of the original size).



On the notebook Cifar 10 - Image Classification on a Raspi with TFLite (which can be run over the Raspi), we can follow the same steps we did with the `mobilenet_v2_1.0_224_quant.tflite`. Below are examples of images using the *General Function for Image Classification* on a Raspi-Zero, as shown in the last section.



```
[PREDICTION]         [Prob]

dog                  : 45.8%
cat                  : 28.7%
bird                 : 9.3%
horse                : 5.5%
truck                : 4.2%

Inference time: 2.8ms
```

```
[PREDICTION]         [Prob]

cat                  : 50.2%
dog                  : 24.9%
frog                 : 14.3%
deer                 : 2.2%
bird                 : 1.9%

Inference time: 2.8ms
```

```
[PREDICTION]         [Prob]

ship                 : 88.7%
airplane             : 9.2%
deer                 : 1.0%
bird                 : 0.2%
automobile           : 0.0%

Inference time: 2.0ms
```

```
[PREDICTION]         [Prob]

automobile           : 99.7%
truck                : 0.3%
airplane             : 0.0%
ship                 : 0.0%
bird                 : 0.0%

Inference time: 2.8ms
```

Installing Picamera2

Picamera2, a Python library for interacting with Raspberry Pi's camera, is based on the *libcamera* camera stack, and the Raspberry Pi foundation maintains it. The Picamera2 library is supported on all Raspberry Pi models, from the Pi Zero to the RPi 5. It is already installed system-wide on the Raspi, but we should make it accessible within the virtual environment.

1. First, activate the virtual environment if it's not already activated:

```
source ~/tflite/bin/activate
```

2. Now, let's create a .pth file in your virtual environment to add the system site-packages path:

```
echo "/usr/lib/python3/dist-packages" > \
 $VIRTUAL_ENV/lib/python3.11/
site-packages/system_site_packages.pth
```

> Note: If your Python version differs, replace `python3.11` with the appropriate version.

3. After creating this file, try importing picamera2 in Python:

```
python3
>>> import picamera2
>>> print(picamera2.__file__)
```

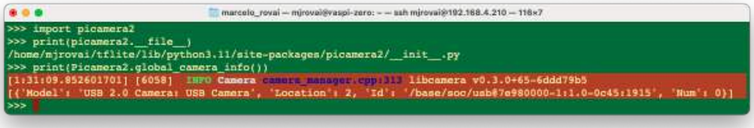
The above code will show the file location of the `picamera2` module itself, proving that the library can be accessed from the environment.

```
/home/mjrovai/tflite/lib/python3.11/site-packages/\
picamera2/__init__.py
```

You can also list the available cameras in the system:

```
>>> print(Picamera2.global_camera_info())
```

In my case, with a USB installed, I got:

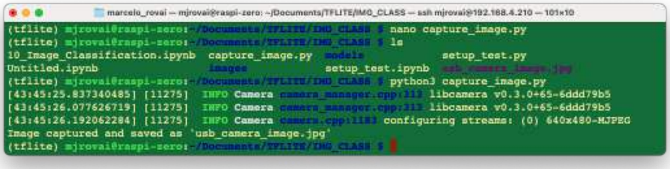

marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 118×7

```
>>> import picamera2
>>> print(picamera2.__file__)
/home/mjrovai/tflite/lib/python3.11/site-packages/picamera2/__init__.py
>>> print(Picamera2.global_camera_info())
[1:31:09.852601701] [6058]  INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5
[{'Model': 'USB 2.0 Camera: USB Camera', 'Location': 2, 'Id': '/base/soc/usb@7e980000-1:1.0-0c45:1915', 'Num': 0}]
>>> 
```

Now that we've confirmed picamera2 is working in the environment with an `index 0`, let's try a simple Python script to capture an image from your USB camera:

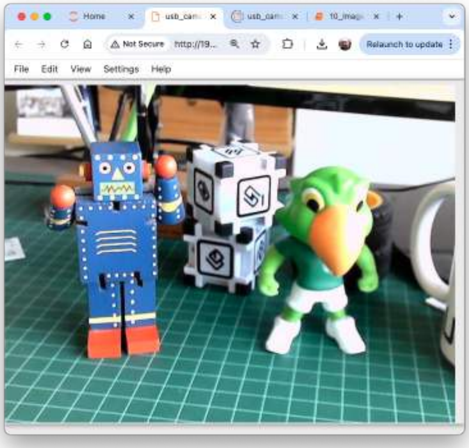
```python
from picamera2 import Picamera2
import time

# Initialize the camera
picam2 = Picamera2() # default is index 0

# Configure the camera
config = picam2.create_still_configuration(
    main={"size": (640, 480)})
picam2.configure(config)

# Start the camera
picam2.start()

# Wait for the camera to warm up
time.sleep(2)

# Capture an image
picam2.capture_file("usb_camera_image.jpg")
print("Image captured and saved as 'usb_camera_image.jpg'")

# Stop the camera
picam2.stop()
```

Use the Nano text editor, the Jupyter Notebook, or any other editor. Save this as a Python script (e.g., `capture_image.py`) and run it. This should capture an image from your camera and save it as "usb_camera_image.jpg" in the same directory as your script.


marcelo_rovai — mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS — ssh mjrovai@192.168.4.210 — 101×10

```
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ nano capture_image.py
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ ls
10_Image_Classification.ipynb  capture_image.py  models              setup_test.py
Untitled.ipynb                 images            setup_test.ipynb    usb_camera_image.jpg
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ python3 capture_image.py
[43:45:25.837340485] [11275]  INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5
[43:45:26.077626719] [11275]  INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5
[43:45:26.192062284] [11275]  INFO Camera camera.cpp:1183 configuring streams: (0) 640x480-MJPEG
Image captured and saved as 'usb_camera_image.jpg'
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ 
```

If the Jupyter is open, you can see the captured image on your computer. Otherwise, transfer the file from the Raspi to your computer.



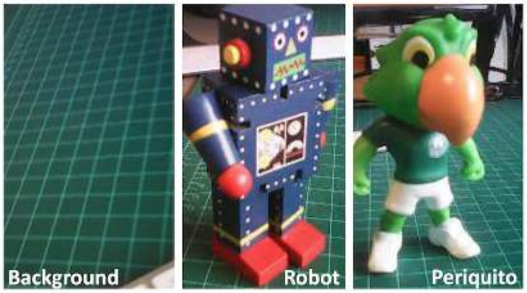
> If you are working with a Raspi-5 with a whole desktop, you can open the file directly on the device.

Image Classification Project

Now, we will develop a complete Image Classification project using the Edge Impulse Studio. As we did with the Movilinet V2, the trained and converted TFLite model will be used for inference.

The Goal

The first step in any ML project is to define its goal. In this case, it is to detect and classify two specific objects present in one image. For this project, we will use two small toys: a robot and a small Brazilian parrot (named Periquito). We will also collect images of a *background* where those two objects are absent.



Data Collection

Once we have defined our Machine Learning project goal, the next and most crucial step is collecting the dataset. We can use a phone for the image capture, but we will use the Raspi here. Let's set up a simple web server on our Raspberry Pi to view the `QVGA (320 x 240)` captured images in a browser.

1. First, let's install Flask, a lightweight web framework for Python:

```
pip3 install flask
```

2. Let's create a new Python script combining image capture with a web server. We'll call it `get_img_data.py`:

```
from flask import Flask, Response, render_template_string,
                  request, redirect, url_for
from picamera2 import Picamera2
import io
import threading
import time
import os
import signal

app = Flask(__name__)

# Global variables
base_dir = "dataset"
picam2 = None
frame = None
frame_lock = threading.Lock()
capture_counts = {}
current_label = None
shutdown_event = threading.Event()

def initialize_camera():
    global picam2
    picam2 = Picamera2()
    config = picam2.create_preview_configuration(
             main={"size": (320, 240)}
    )
    picam2.configure(config)
    picam2.start()
    time.sleep(2)  # Wait for camera to warm up

def get_frame():
    global frame
    while not shutdown_event.is_set():
        stream = io.BytesIO()
        picam2.capture_file(stream, format='jpeg')
        with frame_lock:
            frame = stream.getvalue()
        time.sleep(0.1)  # Adjust as needed for smooth preview

def generate_frames():
    while not shutdown_event.is_set():
        with frame_lock:
            if frame is not None:
                yield (b'--frame\r\n'
                       b'Content-Type: image/jpeg\r\n\r\n' +
                                       frame + b'\r\n')
        time.sleep(0.1)  # Adjust as needed for smooth streaming

def shutdown_server():
    shutdown_event.set()
    if picam2:
        picam2.stop()
    # Give some time for other threads to finish
    time.sleep(2)
    # Send SIGINT to the main process
    os.kill(os.getpid(), signal.SIGINT)

@app.route('/', methods=['GET', 'POST'])
def index():
    global current_label
    if request.method == 'POST':
        current_label = request.form['label']
        if current_label not in capture_counts:
            capture_counts[current_label] = 0
        os.makedirs(os.path.join(base_dir, current_label),
                                 exist_ok=True)
        return redirect(url_for('capture_page'))
    return render_template_string('''
        <!DOCTYPE html>
        <html>
        <head>
            <title>Dataset Capture - Label Entry</title>
        </head>
        <body>
            <h1>Enter Label for Dataset</h1>
            <form method="post">
                <input type="text" name="label" required>
                <input type="submit" value="Start Capture">
            </form>
        </body>
        </html>
    ''')
```

```
@app.route('/capture')
def capture_page():
    return render_template_string('''
        <!DOCTYPE html>
        <html>
        <head>
            <title>Dataset Capture</title>
            <script>
                var shutdownInitiated = false;
                function checkShutdown() {
                    if (!shutdownInitiated) {
                        fetch('/check_shutdown')
                            .then(response => response.json())
                            .then(data => {
                                if (data.shutdown) {
                                    shutdownInitiated = true;
                                    document.getElementById(
                                          'video-feed').src = '';
                                    document.getElementById(
                                          'shutdown-message')
                                    .style.display = 'block';
                                }
                            });
                    }
                }
                setInterval(checkShutdown, 1000); // Check
                                                     every second
            </script>
        </head>
        <body>
            <h1>Dataset Capture</h1>
            <p>Current Label: {{ label }}</p>
            <p>Images captured for this label: {{ capture_count
                                                  }}</p>
            <img id="video-feed" src="{{ url_for('video_feed')
                                         }}" width="640"
            height="480" />
            <div id="shutdown-message" style="display: none;
                                              color: red;">
                Capture process has been stopped.
                You can close this window.
            </div>
            <form action="/capture_image" method="post">
                <input type="submit" value="Capture Image">
            </form>
            <form action="/stop" method="post">
```

```
                <input type="submit" value="Stop Capture"
                style="background-color: #ff6666;">
            </form>
            <form action="/" method="get">
                <input type="submit" value="Change Label"
                style="background-color: #ffff66;">
            </form>
        </body>
        </html>
    ''', label=current_label, capture_count=capture_counts.get(
                                            current_label, 0))

@app.route('/video_feed')
def video_feed():
    return Response(generate_frames(),
                    mimetype='multipart/x-mixed-replace;
                    boundary=frame')

@app.route('/capture_image', methods=['POST'])
def capture_image():
    global capture_counts
    if current_label and not shutdown_event.is_set():
        capture_counts[current_label] += 1
        timestamp = time.strftime("%Y%m%d-%H%M%S")
        filename = f"image_{timestamp}.jpg"
        full_path = os.path.join(base_dir, current_label,
                                 filename)

        picam2.capture_file(full_path)

    return redirect(url_for('capture_page'))

@app.route('/stop', methods=['POST'])
def stop():
    summary = render_template_string('''
        <!DOCTYPE html>
        <html>
        <head>
            <title>Dataset Capture - Stopped</title>
        </head>
        <body>
            <h1>Dataset Capture Stopped</h1>
            <p>The capture process has been stopped.
               You can close this window.</p>
            <p>Summary of captures:</p>
            <ul>
            {% for label, count in capture_counts.items() %}
```

```
                <li>{{ label }}: {{ count }} images</li>
            {% endfor %}
            </ul>
        </body>
        </html>
    ''', capture_counts=capture_counts)

    # Start a new thread to shutdown the server
    threading.Thread(target=shutdown_server).start()

    return summary

@app.route('/check_shutdown')
def check_shutdown():
    return {'shutdown': shutdown_event.is_set()}

if __name__ == '__main__':
    initialize_camera()
    threading.Thread(target=get_frame, daemon=True).start()
    app.run(host='0.0.0.0', port=5000, threaded=True)
```

3. Run this script:

```
    python3 get_img_data.py
```

4. Access the web interface:
    - On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`
    - From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace `<raspberry_pi_ip>` with your Raspberry Pi's IP address). For example: `http://192.168.4.210:5000/`

This Python script creates a web-based interface for capturing and organizing image datasets using a Raspberry Pi and its camera. It's handy for machine learning projects that require labeled image data.

Key Features:

1. **Web Interface**: Accessible from any device on the same network as the Raspberry Pi.
2. **Live Camera Preview**: This shows a real-time feed from the camera.
3. **Labeling System**: Allows users to input labels for different categories of images.
4. **Organized Storage**: Automatically saves images in label-specific subdirectories.

5. **Per-Label Counters**: Keeps track of how many images are captured for each label.
6. **Summary Statistics**: Provides a summary of captured images when stopping the capture process.

Main Components:

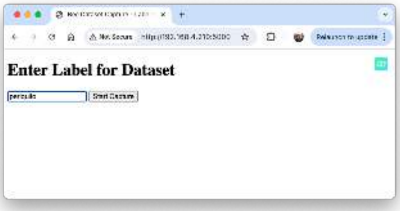
1. **Flask Web Application**: Handles routing and serves the web interface.
2. **Picamera2 Integration**: Controls the Raspberry Pi camera.
3. **Threaded Frame Capture**: Ensures smooth live preview.
4. **File Management**: Organizes captured images into labeled directories.

Key Functions:

- `initialize_camera()`: Sets up the Picamera2 instance.
- `get_frame()`: Continuously captures frames for the live preview.
- `generate_frames()`: Yields frames for the live video feed.
- `shutdown_server()`: Sets the shutdown event, stops the camera, and shuts down the Flask server
- `index()`: Handles the label input page.
- `capture_page()`: Displays the main capture interface.
- `video_feed()`: Shows a live preview to position the camera
- `capture_image()`: Saves an image with the current label.
- `stop()`: Stops the capture process and displays a summary.

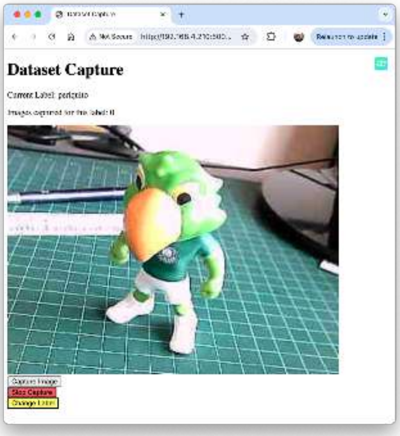
Usage Flow:

1. Start the script on your Raspberry Pi.
2. Access the web interface from a browser.
3. Enter a label for the images you want to capture and press `Start Capture`.



Enter Label for Dataset

4. Use the live preview to position the camera.
5. Click `Capture Image` to save images under the current label.

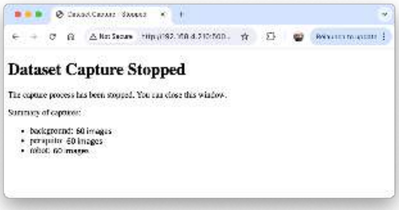


Dataset Capture

Current Label: periquito

Images captured for this label: 0

6. Change labels as needed for different categories, selecting `Change Label`.
7. Click `Stop Capture` when finished to see a summary.



Dataset Capture Stopped

The capture process has been stopped. You can close this window.

Summary of captures:

- background: 60 images
- periquito: 60 images
- robot: 60 images

Technical Notes:

- The script uses threading to handle concurrent frame capture and web serving.
- Images are saved with timestamps in their filenames for uniqueness.
- The web interface is responsive and can be accessed from mobile devices.

Customization Possibilities:

- Adjust image resolution in the `initialize_camera()` function. Here we used QVGA (320×240).
- Modify the HTML templates for a different look and feel.
- Add additional image processing or analysis steps in the `capture_image()` function.

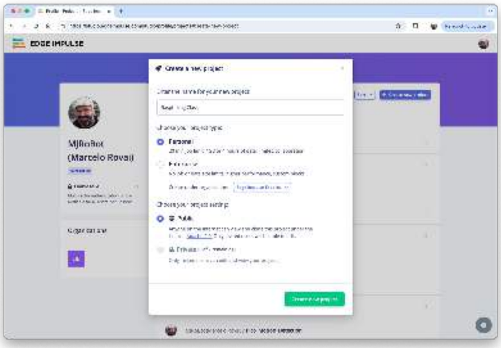
Number of samples on Dataset:

Get around 60 images from each category (`periquito`, `robot` and `background`). Try to capture different angles, backgrounds, and light conditions. On the Raspi, we will end with a folder named `dataset`, witch contains 3 sub-folders *periquito*, *robot*, and *background*. one for each class of images.

You can use `Filezilla` to transfer the created dataset to your main computer.

Training the model with Edge Impulse Studio

We will use the Edge Impulse Studio to train our model. Go to the Edge Impulse Page, enter your account credentials, and create a new project:



> Here, you can clone a similar project: Raspi - Img Class.

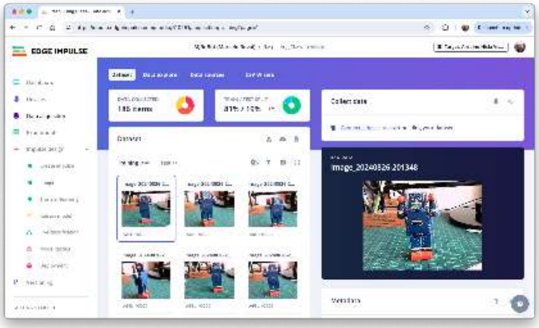
Dataset

We will walk through four main steps using the EI Studio (or Studio). These steps are crucial in preparing our model for use on the Raspi: Dataset, Impulse, Tests, and Deploy (on the Edge Device, in this case, the Raspi).

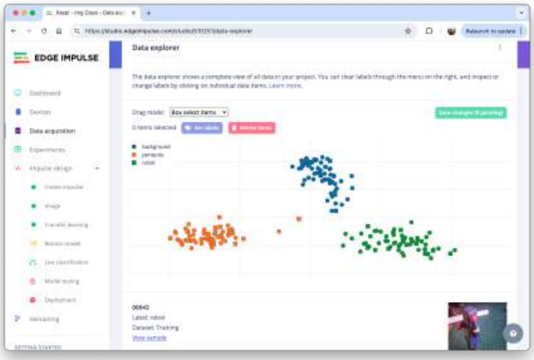
> Regarding the Dataset, it is essential to point out that our Original Dataset, captured with the Raspi, will be split into *Training*, *Validation*, and *Test*. The Test Set will be separated from the beginning and reserved for use only in the Test phase after training. The Validation Set will be used during training.

On Studio, follow the steps to upload the captured data:

1. Go to the `Data acquisition` tab, and in the `UPLOAD DATA` section, upload the files from your computer in the chosen categories.
2. Leave to the Studio the splitting of the original dataset into *train and test* and choose the label about
3. Repeat the procedure for all three classes. At the end, you should see your “raw data” in the Studio:



The Studio allows you to explore your data, showing a complete view of all the data in your project. You can clear, inspect, or change labels by clicking on individual data items. In our case, a straightforward project, the data seems OK.

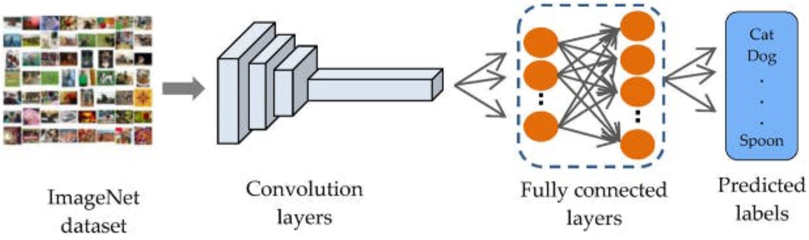


The Impulse Design

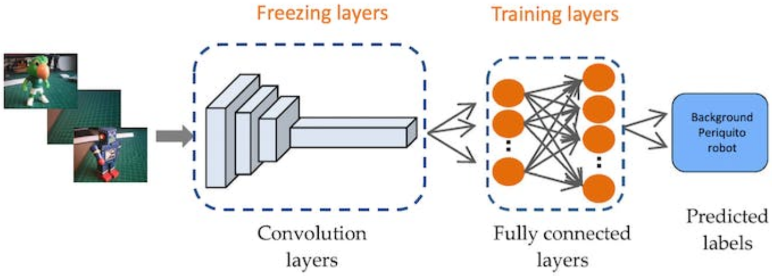
In this phase, we should define how to:

- Pre-process our data, which consists of resizing the individual images and determining the `color depth` to use (be it RGB or Grayscale) and
- Specify a Model. In this case, it will be the `Transfer Learning (Images)` to fine-tune a pre-trained MobileNet V2 image classification model on our data. This method performs well even with relatively small image datasets (around 180 images in our case).

Transfer Learning with MobileNet offers a streamlined approach to model training, which is especially beneficial for resource-constrained environments and projects with limited labeled data. MobileNet, known for its lightweight architecture, is a pre-trained model that has already learned valuable features from a large dataset (ImageNet).



By leveraging these learned features, we can train a new model for your specific task with fewer data and computational resources and achieve competitive accuracy.



This approach significantly reduces training time and computational cost, making it ideal for quick prototyping and deployment on embedded devices where efficiency is paramount.

Go to the Impulse Design Tab and create the *impulse*, defining an image size of 160 × 160 and squashing them (squared form, without cropping). Select Image and Transfer Learning blocks. Save the Impulse.

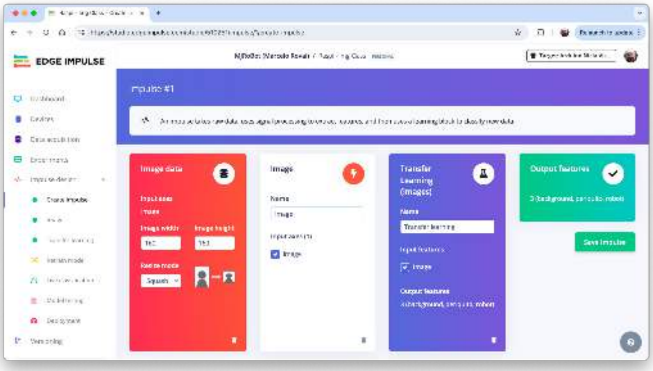


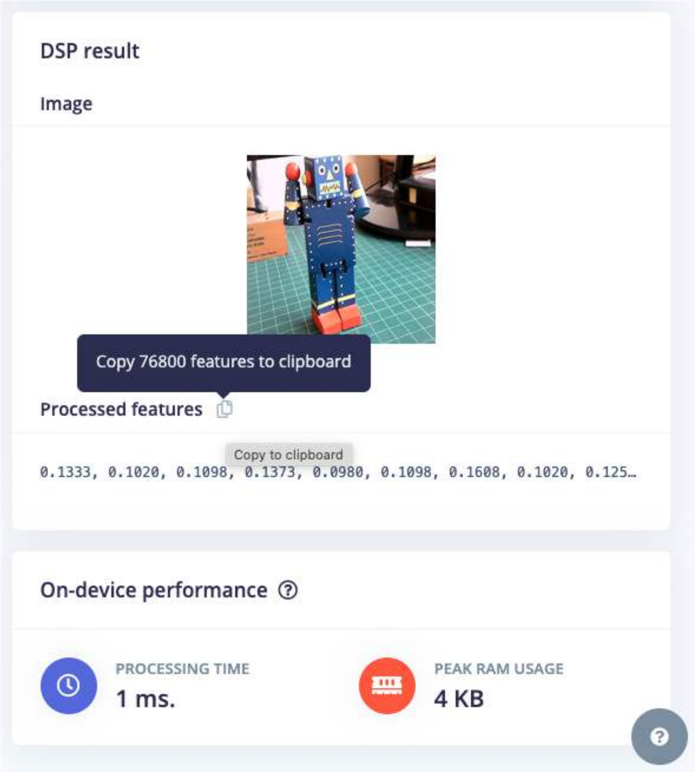
Image width: 160
Image height: 160
Resize mode: Squash

Name: Image
Input axes: ☑ Image

Name: Transfer learning
Input features: ☑ Image
Output features: 3 (background, periquito, robot)

Image Pre-Processing

All the input QVGA/RGB565 images will be converted to 76,800 features (160 × 160 × 3).



DSP result

Image

Copy 76800 features to clipboard

Processed features

Copy to clipboard

```
0.1333, 0.1020, 0.1098, 0.1373, 0.0980, 0.1098, 0.1608, 0.1020, 0.125…
```

On-device performance

PROCESSING TIME
1 ms.

PEAK RAM USAGE
4 KB

Press `Save parameters` and select `Generate features` in the next tab.

Model Design

MobileNet is a family of efficient convolutional neural networks designed for mobile and embedded vision applications. The key features of MobileNet are:

1. Lightweight: Optimized for mobile devices and embedded systems with limited computational resources.
2. Speed: Fast inference times, suitable for real-time applications.
3. Accuracy: Maintains good accuracy despite its compact size.

MobileNetV2, introduced in 2018, improves the original MobileNet architecture. Key features include:

1. Inverted Residuals: Inverted residual structures are used where shortcut connections are made between thin bottleneck layers.
2. Linear Bottlenecks: Removes non-linearities in the narrow layers to prevent the destruction of information.

3. Depth-wise Separable Convolutions: Continues to use this efficient operation from MobileNetV1.

In our project, we will do a `Transfer Learning` with the `MobileNetV2 160x160 1.0`, which means that the images used for training (and future inference) should have an *input Size* of 160 × 160 pixels and a *Width Multiplier* of 1.0 (full width, not reduced). This configuration balances between model size, speed, and accuracy.

Model Training

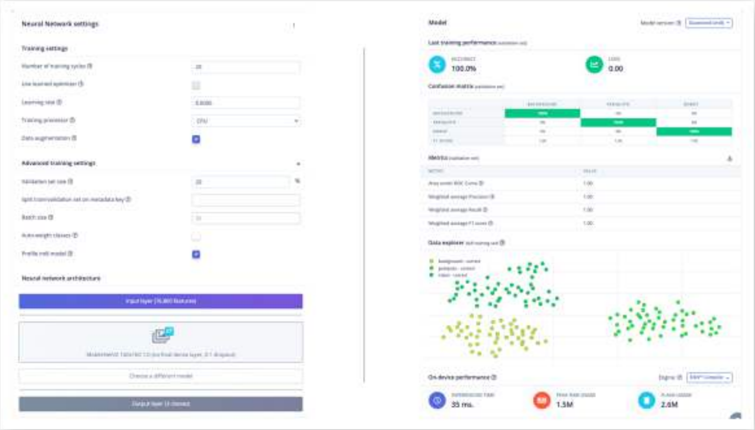
Another valuable deep learning technique is **Data Augmentation**. Data augmentation improves the accuracy of machine learning models by creating additional artificial data. A data augmentation system makes small, random changes to the training data during the training process (such as flipping, cropping, or rotating the images).

Looking under the hood, here you can see how Edge Impulse implements a data Augmentation policy on your data:

```python
# Implements the data augmentation policy
def augment_image(image, label):
    # Flips the image randomly
    image = tf.image.random_flip_left_right(image)

    # Increase the image size, then randomly crop it down to
    # the original dimensions
    resize_factor = random.uniform(1, 1.2)
    new_height = math.floor(resize_factor * INPUT_SHAPE[0])
    new_width = math.floor(resize_factor * INPUT_SHAPE[1])
    image = tf.image.resize_with_crop_or_pad(image, new_height,
                                             new_width)
    image = tf.image.random_crop(image, size=INPUT_SHAPE)

    # Vary the brightness of the image
    image = tf.image.random_brightness(image, max_delta=0.2)

    return image, label
```

Exposure to these variations during training can help prevent your model from taking shortcuts by “memorizing” superficial clues in your training data, meaning it may better reflect the deep underlying patterns in your dataset.

The final dense layer of our model will have 0 neurons with a 10% dropout for overfitting prevention. Here is the Training result:



Neural Network settings

Training settings

Advanced training settings

Neural network architecture

Model

Last training performance

ACCURACY 100.0%

LOSS 0.00

Confusion matrix

Metrics

Data explorer

On-device performance

35 ms.

1.5M

2.6M

The result is excellent, with a reasonable 35 ms of latency (for a Raspi-4), which should result in around 30 fps (frames per second) during inference. A Raspi-Zero should be slower, and the Raspi-5, faster.

Trading off: Accuracy versus speed

If faster inference is needed, we should train the model using smaller alphas (0.35, 0.5, and 0.75) or even reduce the image input size, trading with accuracy. However, reducing the input image size and decreasing the alpha (width multiplier) can speed up inference for MobileNet V2, but they have different trade-offs. Let's compare:

1. Reducing Image Input Size:

Pros:

- Significantly reduces the computational cost across all layers.
- Decreases memory usage.
- It often provides a substantial speed boost.

Cons:

- It may reduce the model's ability to detect small features or fine details.
- It can significantly impact accuracy, especially for tasks requiring fine-grained recognition.

2. Reducing Alpha (Width Multiplier):

Pros:

- Reduces the number of parameters and computations in the model.
- Maintains the original input resolution, potentially preserving more detail.
- It can provide a good balance between speed and accuracy.

Cons:

- It may not speed up inference as dramatically as reducing input size.

- It can reduce the model's capacity to learn complex features.

Comparison:

1. Speed Impact:
   - Reducing input size often provides a more substantial speed boost because it reduces computations quadratically (halving both width and height reduces computations by about 75%).
   - Reducing alpha provides a more linear reduction in computations.
2. Accuracy Impact:
   - Reducing input size can severely impact accuracy, especially when detecting small objects or fine details.
   - Reducing alpha tends to have a more gradual impact on accuracy.
3. Model Architecture:
   - Changing input size doesn't alter the model's architecture.
   - Changing alpha modifies the model's structure by reducing the number of channels in each layer.

Recommendation:

1. If our application doesn't require detecting tiny details and can tolerate some loss in accuracy, reducing the input size is often the most effective way to speed up inference.
2. Reducing alpha might be preferable if maintaining the ability to detect fine details is crucial or if you need a more balanced trade-off between speed and accuracy.
3. For best results, you might want to experiment with both:
   - Try MobileNet V2 with input sizes like 160×160 or 92×92
   - Experiment with alpha values like 1.0, 0.75, 0.5 or 0.35.
4. Always benchmark the different configurations on your specific hardware and with your particular dataset to find the optimal balance for your use case.

   Remember, the best choice depends on your specific requirements for accuracy, speed, and the nature of the images you're working with. It's often worth experimenting with combinations to find the optimal configuration for your particular use case.

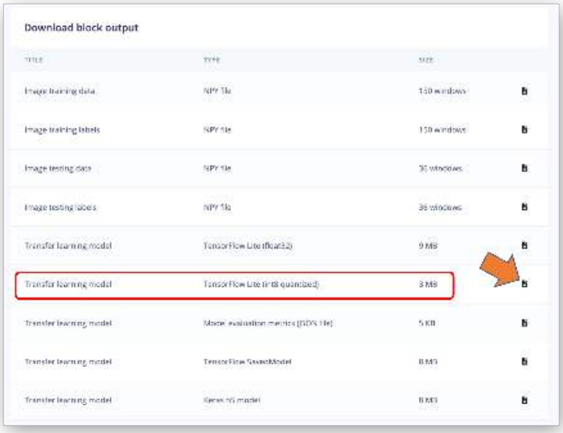
Model Testing

Now, you should take the data set aside at the start of the project and run the trained model using it as input. Again, the result is excellent (92.22%).

Deploying the model

As we did in the previous section, we can deploy the trained model as .tflite and use Raspi to run it using Python.

On the `Dashboard` tab, go to Transfer learning model (int8 quantized) and click on the download icon:



Download block output

TITLE	TYPE	SIZE
Image training data	NPY file	150 windows
Image training labels	NPY file	150 windows
Image testing data	NPY file	36 windows
Image testing labels	NPY file	36 windows
Transfer learning model	TensorFlow Lite (float32)	9 MB
Transfer learning model	TensorFlow Lite (int8 quantized)	3 MB
Transfer learning model	Model evaluation metrics (JSON file)	5 KB
Transfer learning model	TensorFlow SavedModel	8 MB
Transfer learning model	Keras h5 model	8 MB

> Let's also download the float32 version for comparison

Transfer the model from your computer to the Raspi (./models), for example, using FileZilla. Also, capture some images for inference (./images).

Import the needed libraries:

```python
import time
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tflite_runtime.interpreter as tflite
```

Define the paths and labels:

```python
img_path = "./images/robot.jpg"
model_path = "./models/ei-raspi-img-class-int8-quantized-\
                model.tflite"
labels = ['background', 'periquito', 'robot']
```

> Note that the models trained on the Edge Impulse Studio will output values with index 0, 1, 2, etc., where the actual labels will follow an alphabetic order.

Load the model, allocate the tensors, and get the input and output tensor details:

```python
# Load the TFLite model
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
```

```
# Get input and output tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

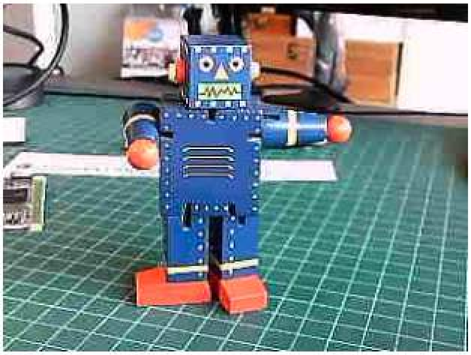
One important difference to note is that the `dtype` of the input details of the model is now `int8`, which means that the input values go from –128 to +127, while each pixel of our image goes from 0 to 255. This means that we should pre-process the image to match it. We can check here:

```
input_dtype = input_details[0]['dtype']
input_dtype
```

```
numpy.int8
```

So, let's open the image and show it:

```
img = Image.open(img_path)
plt.figure(figsize=(4, 4))
plt.imshow(img)
plt.axis('off')
plt.show()
```



And perform the pre-processing:

```
scale, zero_point = input_details[0]['quantization']
img = img.resize((input_details[0]['shape'][1],
                  input_details[0]['shape'][2]))
img_array = np.array(img, dtype=np.float32) / 255.0
img_array = (
    (img_array / scale + zero_point)
    .clip(-128, 127)
    .astype(np.int8)
)
input_data = np.expand_dims(img_array, axis=0)
```

Checking the input data, we can verify that the input tensor is compatible with what is expected by the model:

```
input_data.shape, input_data.dtype
```

```
((1, 160, 160, 3), dtype('int8'))
```

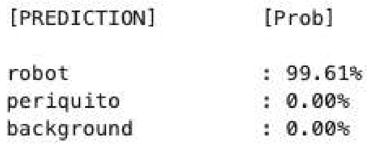
Now, it is time to perform the inference. Let's also calculate the latency of the model:

```
# Inference on Raspi-Zero
start_time = time.time()
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
end_time = time.time()
inference_time = (end_time - start_time) * 1000 # Convert
                                                # to milliseconds
print ("Inference time: {:.1f}ms".format(inference_time))
```

The model will take around 125ms to perform the inference in the Raspi-Zero, which is 3 to 4 times longer than a Raspi-5.

Now, we can get the output labels and probabilities. It is also important to note that the model trained on the Edge Impulse Studio has a softmax in its output (different from the original Movilenet V2), and we should use the model's raw output as the "probabilities."

```
# Obtain results and map them to the classes
predictions = interpreter.get_tensor(output_details[0]
                                     ['index'])[0]

# Get indices of the top k results
top_k_results=3
top_k_indices = np.argsort(predictions)[::-1][:top_k_results]

# Get quantization parameters
scale, zero_point = output_details[0]['quantization']

# Dequantize the output
dequantized_output = (predictions.astype(np.float32) -
                      zero_point) * scale
probabilities = dequantized_output

print("\n\t[PREDICTION]        [Prob]\n")
for i in range(top_k_results):
    print("\t{:20}: {:.2f}%".format(
        labels[top_k_indices[i]],
        probabilities[top_k_indices[i]] * 100))
```



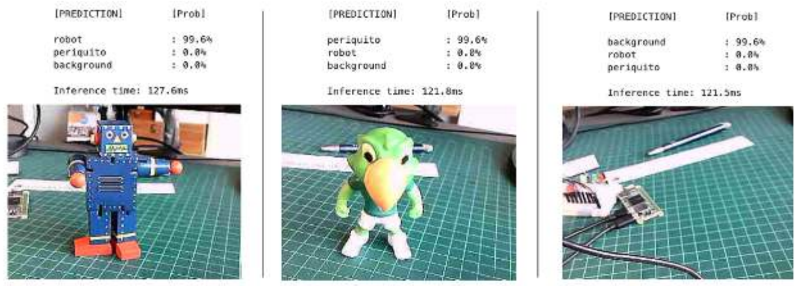
```
[PREDICTION]        [Prob]

robot               : 99.61%
periquito           : 0.00%
background          : 0.00%
```

Let's modify the function created before so that we can handle different type of models:

```python
def image_classification(img_path, model_path, labels,
                         top_k_results=3, apply_softmax=False):
    # Load the image
    img = Image.open(img_path)
    plt.figure(figsize=(4, 4))
    plt.imshow(img)
    plt.axis('off')

    # Load the TFLite model
    interpreter = tflite.Interpreter(model_path=model_path)
    interpreter.allocate_tensors()

    # Get input and output tensors
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()

    # Preprocess
    img = img.resize((input_details[0]['shape'][1],
                      input_details[0]['shape'][2]))

    input_dtype = input_details[0]['dtype']

    if input_dtype == np.uint8:
        input_data = np.expand_dims(np.array(img), axis=0)
    elif input_dtype == np.int8:
        scale, zero_point = input_details[0]['quantization']
        img_array = np.array(img, dtype=np.float32) / 255.0
        img_array = (
            img_array / scale
            + zero_point
        ).clip(-128, 127).astype(np.int8)
        input_data = np.expand_dims(img_array, axis=0)
    else:  # float32
        input_data = np.expand_dims(
            np.array(img, dtype=np.float32),
            axis=0
        ) / 255.0
```

```
# Inference on Raspi-Zero
start_time = time.time()
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
end_time = time.time()
inference_time = (end_time -
                    start_time
                  ) * 1000 # Convert to milliseconds

# Obtain results
predictions = interpreter.get_tensor(output_details[0]
                                     ['index'])[0]

# Get indices of the top k results
top_k_indices = np.argsort(predictions)[::-1][:top_k_results]

# Handle output based on type
output_dtype = output_details[0]['dtype']
if output_dtype in [np.int8, np.uint8]:
    # Dequantize the output
    scale, zero_point = output_details[0]['quantization']
    predictions = (predictions.astype(np.float32) -
                   zero_point) * scale

if apply_softmax:
    # Apply softmax
    exp_preds = np.exp(predictions - np.max(predictions))
    probabilities = exp_preds / np.sum(exp_preds)
else:
    probabilities = predictions

print("\n\t[PREDICTION]        [Prob]\n")
for i in range(top_k_results):
    print("\t{:20}: {:.1f}%".format(
        labels[top_k_indices[i]],
        probabilities[top_k_indices[i]] * 100))
print ("\n\tInference time: {:.1f}ms".format(inference_time))
```

And test it with different images and the int8 quantized model (**160x160 alpha =1.0**).

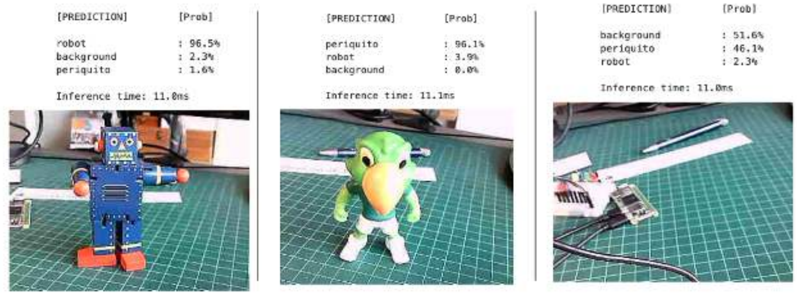


```
[PREDICTION]          [Prob]

robot                 : 99.6%
periquito             : 0.0%
background            : 0.0%

Inference time: 127.6ms
```

```
[PREDICTION]          [Prob]

periquito             : 99.6%
robot                 : 0.0%
background            : 0.0%

Inference time: 121.8ms
```

```
[PREDICTION]          [Prob]

background            : 99.6%
robot                 : 0.0%
periquito             : 0.0%

Inference time: 121.5ms
```

Let's download a smaller model, such as the one trained for the Nicla Vision Lab (int8 quantized model, 96x96, alpha = 0.1), as a test. We can use the same function:



```
[PREDICTION]          [Prob]

robot                 : 96.5%
background            : 2.3%
periquito             : 1.6%

Inference time: 11.0ms
```

```
[PREDICTION]          [Prob]

periquito             : 96.1%
robot                 : 3.9%
background            : 0.0%

Inference time: 11.1ms
```

```
[PREDICTION]          [Prob]

background            : 51.6%
periquito             : 46.1%
robot                 : 2.3%

Inference time: 11.0ms
```

The model lost some accuracy, but it is still OK once our model does not look for many details. Regarding latency, we are around **ten times faster** on the Raspi-Zero.

Live Image Classification

Let's develop an app to capture images with the USB camera in real time, showing its classification.

Using the nano on the terminal, save the code below, such as `img_class_live_infer.py`.

```
from flask import Flask, Response, render_template_string,
                  request, jsonify
from picamera2 import Picamera2
import io
import threading
import time
import numpy as np
from PIL import Image
import tflite_runtime.interpreter as tflite
from queue import Queue
```

```
app = Flask(__name__)

# Global variables
picam2 = None
frame = None
frame_lock = threading.Lock()
is_classifying = False
confidence_threshold = 0.8
model_path = "./models/ei-raspi-img-class-int8-quantized-\
                model.tflite"
labels = ['background', 'periquito', 'robot']
interpreter = None
classification_queue = Queue(maxsize=1)

def initialize_camera():
    global picam2
    picam2 = Picamera2()
    config = picam2.create_preview_configuration(
        main={"size": (320, 240)}
    )
    picam2.configure(config)
    picam2.start()
    time.sleep(2)  # Wait for camera to warm up

def get_frame():
    global frame
    while True:
        stream = io.BytesIO()
        picam2.capture_file(stream, format='jpeg')
        with frame_lock:
            frame = stream.getvalue()
        time.sleep(0.1)  # Capture frames more frequently

def generate_frames():
    while True:
        with frame_lock:
            if frame is not None:
                yield (
                   b'--frame\r\n'
                   b'Content-Type: image/jpeg\r\n\r\n'
                   + frame + b'\r\n'
                )
        time.sleep(0.1)

def load_model():
    global interpreter
    if interpreter is None:
        interpreter = tflite.Interpreter(model_path=model_path)
        interpreter.allocate_tensors()
    return interpreter

def classify_image(img, interpreter):
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()

    img = img.resize((input_details[0]['shape'][1],
                      input_details[0]['shape'][2]))
    input_data = np.expand_dims(np.array(img), axis=0)\
                             .astype(input_details[0]['dtype'])

    interpreter.set_tensor(input_details[0]['index'], input_data)
    interpreter.invoke()

    predictions = interpreter.get_tensor(output_details[0]
                                         ['index'])[0]
    # Handle output based on type
    output_dtype = output_details[0]['dtype']
    if output_dtype in [np.int8, np.uint8]:
        # Dequantize the output
        scale, zero_point = output_details[0]['quantization']
        predictions = (predictions.astype(np.float32) -
                       zero_point) * scale
    return predictions

def classification_worker():
    interpreter = load_model()
    while True:
        if is_classifying:
            with frame_lock:
                if frame is not None:
                    img = Image.open(io.BytesIO(frame))
            predictions = classify_image(img, interpreter)
            max_prob = np.max(predictions)
            if max_prob >= confidence_threshold:
                label = labels[np.argmax(predictions)]
            else:
                label = 'Uncertain'
            classification_queue.put({
                 'label': label,
                 'probability': float(max_prob)
            })
        time.sleep(0.1)  # Adjust based on your needs

@app.route('/')
```

```
def index():
   return render_template_string('''
      <!DOCTYPE html>
      <html>
      <head>
          <title>Image Classification</title>
          <script
            src="https://code.jquery.com/jquery-3.6.0.min.js">
          </script>
          <script>
              function startClassification() {
                  $.post('/start');
                  $('#startBtn').prop('disabled', true);
                  $('#stopBtn').prop('disabled', false);
              }
              function stopClassification() {
                  $.post('/stop');
                  $('#startBtn').prop('disabled', false);
                  $('#stopBtn').prop('disabled', true);
              }
              function updateConfidence() {
                  var confidence = $('#confidence').val();
                  $.post('/update_confidence',
                         {confidence: confidence}
                        );
              }
              function updateClassification() {
                  $.get('/get_classification', function(data) {
                    $('#classification').text(data.label + ': '
                    + data.probability.toFixed(2));
                  });
              }
              $(document).ready(function() {
                  setInterval(updateClassification, 100);
                  // Update every 100ms
              });
          </script>
      </head>
      <body>
          <h1>Image Classification</h1>
          <img src="{{ url_for('video_feed') }}"
               width="640"
               height="480" />

          <br>
          <button id="startBtn"
                  onclick="startClassification()">
            Start Classification
          </button>

          <button id="stopBtn"
                  onclick="stopClassification()"
                  disabled>
            Stop Classification
          </button>

          <br>
          <label for="confidence">Confidence Threshold:</label>
          <input type="number"
                 id="confidence"
                 name="confidence"
                 min="0" max="1"
                 step="0.1"
                 value="0.8"
                 onchange="updateConfidence()" />

          <br>
          <div id="classification">
             Waiting for classification...
          </div>

      </body>
      </html>
   ''')

@app.route('/video_feed')
def video_feed():
    return Response(
       generate_frames(),
       mimetype='multipart/x-mixed-replace; boundary=frame'
    )

@app.route('/start', methods=['POST'])
def start_classification():
    global is_classifying
    is_classifying = True
    return '', 204

@app.route('/stop', methods=['POST'])
def stop_classification():
    global is_classifying
    is_classifying = False
    return '', 204
```

```
@app.route('/update_confidence', methods=['POST'])
def update_confidence():
    global confidence_threshold
    confidence_threshold = float(request.form['confidence'])
    return '', 204

@app.route('/get_classification')
def get_classification():
    if not is_classifying:
        return jsonify({'label': 'Not classifying',
                       'probability': 0})
    try:
        result = classification_queue.get_nowait()
    except Queue.Empty:
        result = {'label': 'Processing', 'probability': 0}
    return jsonify(result)

if __name__ == '__main__':
    initialize_camera()
    threading.Thread(target=get_frame, daemon=True).start()
    threading.Thread(target=classification_worker,
                     daemon=True).start()
    app.run(host='0.0.0.0', port=5000, threaded=True)
```

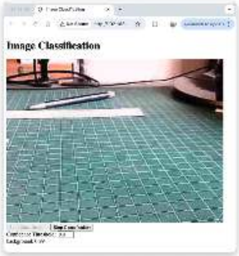
On the terminal, run:

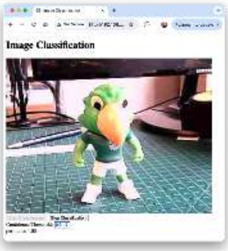
```
python3 img_class_live_infer.py
```

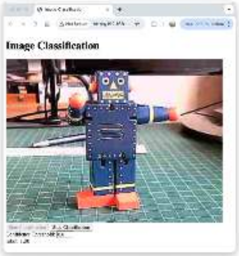
And access the web interface:

- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`
- From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace `<raspberry_pi_ip>` with your Raspberry Pi's IP address). For example: `http://192.168.4.210:5000/`

Here are some screenshots of the app running on an external desktop







Here, you can see the app running on the YouTube:
https://www.youtube.com/watch?v=o1QsQrpCMw4
The code creates a web application for real-time image classification using a Raspberry Pi, its camera module, and a TensorFlow Lite model. The application uses Flask to serve a web interface where is possible to view the camera feed and see live classification results.

Key Components:

1. **Flask Web Application**: Serves the user interface and handles requests.
2. **PiCamera2**: Captures images from the Raspberry Pi camera module.
3. **TensorFlow Lite**: Runs the image classification model.
4. **Threading**: Manages concurrent operations for smooth performance.

Main Features:

- Live camera feed display
- Real-time image classification
- Adjustable confidence threshold
- Start/Stop classification on demand

Code Structure:

1. **Imports and Setup**:
   - Flask for web application
   - PiCamera2 for camera control
   - TensorFlow Lite for inference
   - Threading and Queue for concurrent operations
2. **Global Variables**:
   - Camera and frame management
   - Classification control
   - Model and label information
3. **Camera Functions**:
   - `initialize_camera()`: Sets up the PiCamera2
   - `get_frame()`: Continuously captures frames
   - `generate_frames()`: Yields frames for the web feed
4. **Model Functions**:
   - `load_model()`: Loads the TFLite model
   - `classify_image()`: Performs inference on a single image
5. **Classification Worker**:
   - Runs in a separate thread
   - Continuously classifies frames when active

- Updates a queue with the latest results

6. **Flask Routes**:
   - `/`: Serves the main HTML page
   - `/video_feed`: Streams the camera feed
   - `/start` and `/stop`: Controls classification
   - `/update_confidence`: Adjusts the confidence threshold
   - `/get_classification`: Returns the latest classification result
7. **HTML Template**:
   - Displays camera feed and classification results
   - Provides controls for starting/stopping and adjusting settings
8. **Main Execution**:
   - Initializes camera and starts necessary threads
   - Runs the Flask application

Key Concepts:

1. **Concurrent Operations**: Using threads to handle camera capture and classification separately from the web server.
2. **Real-time Updates**: Frequent updates to the classification results without page reloads.
3. **Model Reuse**: Loading the TFLite model once and reusing it for efficiency.
4. **Flexible Configuration**: Allowing users to adjust the confidence threshold on the fly.

Usage:

1. Ensure all dependencies are installed.
2. Run the script on a Raspberry Pi with a camera module.
3. Access the web interface from a browser using the Raspberry Pi's IP address.
4. Start classification and adjust settings as needed.

Conclusion:

Image classification has emerged as a powerful and versatile application of machine learning, with significant implications for various fields, from healthcare to environmental monitoring. This chapter has demonstrated how to implement a robust image classification system on edge devices like the Raspi-Zero and Raspi-5, showcasing the potential for real-time, on-device intelligence.

We've explored the entire pipeline of an image classification project, from data collection and model training using Edge Impulse Studio to deploying and running inferences on a Raspi. The process highlighted several key points:

1. The importance of proper data collection and preprocessing for training effective models.
2. The power of transfer learning, allowing us to leverage pre-trained models like MobileNet V2 for efficient training with limited data.
3. The trade-offs between model accuracy and inference speed, especially crucial for edge devices.
4. The implementation of real-time classification using a web-based interface, demonstrating practical applications.

The ability to run these models on edge devices like the Raspi opens up numerous possibilities for IoT applications, autonomous systems, and real-time monitoring solutions. It allows for reduced latency, improved privacy, and operation in environments with limited connectivity.

As we've seen, even with the computational constraints of edge devices, it's possible to achieve impressive results in terms of both accuracy and speed. The flexibility to adjust model parameters, such as input size and alpha values, allows for fine-tuning to meet specific project requirements.

Looking forward, the field of edge AI and image classification continues to evolve rapidly. Advances in model compression techniques, hardware acceleration, and more efficient neural network architectures promise to further expand the capabilities of edge devices in computer vision tasks.

This project serves as a foundation for more complex computer vision applications and encourages further exploration into the exciting world of edge AI and IoT. Whether it's for industrial automation, smart home applications, or environmental monitoring, the skills and concepts covered here provide a solid starting point for a wide range of innovative projects.

Resources

- Dataset Example
- Setup Test Notebook on a Raspi
- Image Classification Notebook on a Raspi
- CNN to classify Cifar-10 dataset at CoLab
- Cifar 10 - Image Classification on a Raspi
- Python Scripts
- Edge Impulse Project

Object Detection

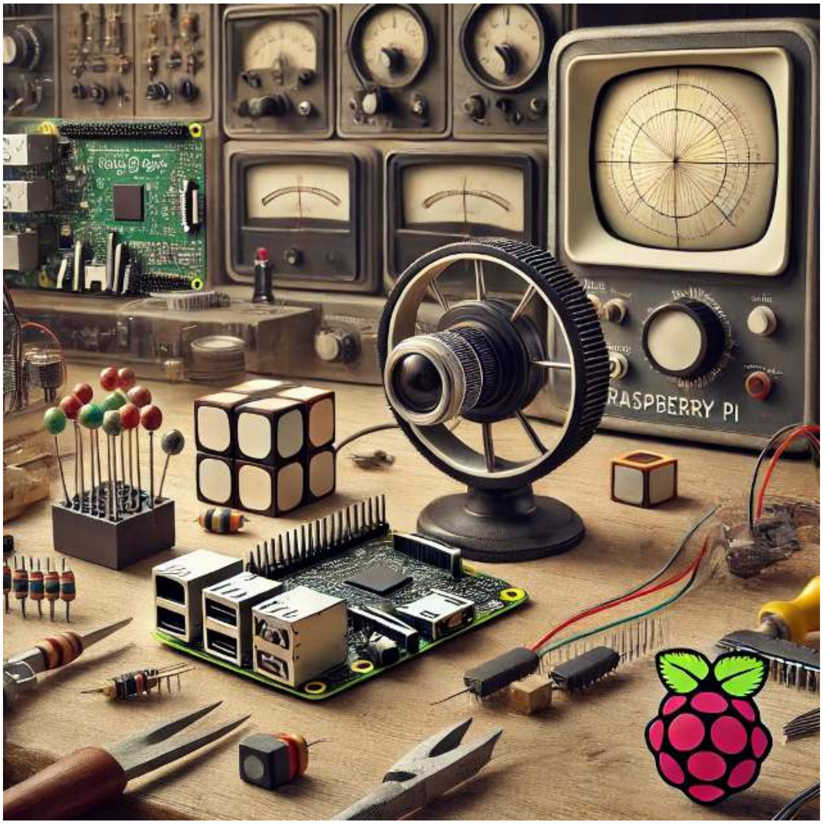


Figure 20.19: *DALL·E prompt - A cover image for an 'Object Detection' chapter in a Raspberry Pi tutorial, designed in the same vintage 1950s electronics lab style as previous covers. The scene should prominently feature wheels and cubes, similar to those provided by the user, placed on a workbench in the foreground. A Raspberry Pi with a connected camera module should be capturing an image of these objects. Surround the scene with classic lab tools like soldering irons, resistors, and wires. The lab background should include vintage equipment like oscilloscopes and tube radios, maintaining the detailed and nostalgic feel of the era. No text or logos should be included.*

Overview

Building upon our exploration of image classification, we now turn our attention to a more advanced computer vision task: object detection. While image classification assigns a single label to an entire image, object detection goes further by identifying and locating multiple objects within a single image. This capability opens up many new applications and challenges, particularly in edge computing and IoT devices like the Raspberry Pi.

Object detection combines the tasks of classification and localization. It not only determines what objects are present in an image but also pinpoints their locations by, for example, drawing bounding boxes around them. This added complexity makes object detection a more powerful tool for understanding visual scenes, but it also requires more sophisticated models and training techniques.

In edge AI, where we work with constrained computational resources, implementing efficient object detection models becomes crucial. The challenges we faced with image classification—balancing model size, inference speed, and accuracy—are amplified in object detection. However, the rewards are also more significant, as object detection enables more nuanced and detailed visual data analysis.

Some applications of object detection on edge devices include:

1. Surveillance and security systems
2. Autonomous vehicles and drones
3. Industrial quality control
4. Wildlife monitoring
5. Augmented reality applications

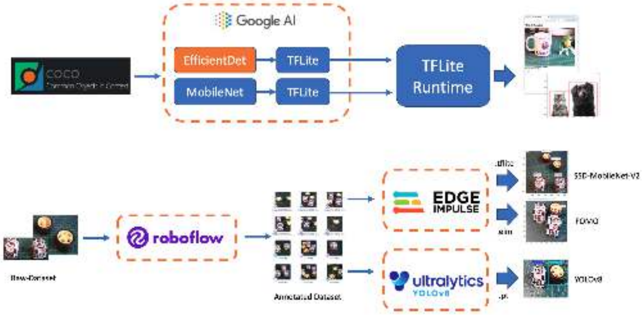
As we put our hands into object detection, we'll build upon the concepts and techniques we explored in image classification. We'll examine popular object detection architectures designed for efficiency, such as:

- Single Stage Detectors, such as MobileNet and EfficientDet,
- FOMO (Faster Objects, More Objects), and
- YOLO (You Only Look Once).

> To learn more about object detection models, follow the tutorial A Gentle Introduction to Object Recognition With Deep Learning.

We will explore those object detection models using

- TensorFlow Lite Runtime (now changed to LiteRT),
- Edge Impulse Linux Python SDK and
- Ultralitics



Throughout this lab, we'll cover the fundamentals of object detection and how it differs from image classification. We'll also learn how to train, fine-tune, test, optimize, and deploy popular object detection architectures using a dataset created from scratch.

Object Detection Fundamentals

Object detection builds upon the foundations of image classification but extends its capabilities significantly. To understand object detection, it's crucial first to recognize its key differences from image classification:

Image Classification vs. Object Detection

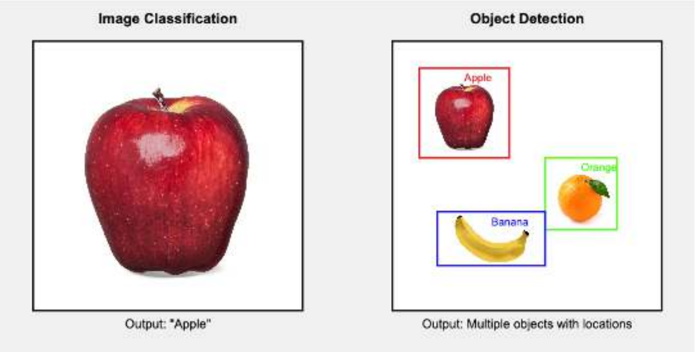
Image Classification:

- Assigns a single label to an entire image
- Answers the question: "What is this image's primary object or scene?"
- Outputs a single class prediction for the whole image

Object Detection:

- Identifies and locates multiple objects within an image
- Answers the questions: "What objects are in this image, and where are they located?"
- Outputs multiple predictions, each consisting of a class label and a bounding box

To visualize this difference, let's consider an example:



This diagram illustrates the critical difference: image classification provides a single label for the entire image, while object detection identifies multiple objects, their classes, and their locations within the image.

Key Components of Object Detection

Object detection systems typically consist of two main components:

1. Object Localization: This component identifies where objects are located in the image. It typically outputs bounding boxes, rectangular regions encompassing each detected object.

2. Object Classification: This component determines the class or category of each detected object, similar to image classification but applied to each localized region.

Challenges in Object Detection

Object detection presents several challenges beyond those of image classification:

- Multiple objects: An image may contain multiple objects of various classes, sizes, and positions.
- Varying scales: Objects can appear at different sizes within the image.
- Occlusion: Objects may be partially hidden or overlapping.
- Background clutter: Distinguishing objects from complex backgrounds can be challenging.
- Real-time performance: Many applications require fast inference times, especially on edge devices.

Approaches to Object Detection

There are two main approaches to object detection:

1. Two-stage detectors: These first propose regions of interest and then classify each region. Examples include R-CNN and its variants (Fast R-CNN, Faster R-CNN).
2. Single-stage detectors: These predict bounding boxes (or centroids) and class probabilities in one forward pass of the network. Examples include YOLO (You Only Look Once), EfficientDet, SSD (Single Shot Detector), and FOMO (Faster Objects, More Objects). These are often faster and more suitable for edge devices like Raspberry Pi.

Evaluation Metrics

Object detection uses different metrics compared to image classification:

- **Intersection over Union (IoU)**: Measures the overlap between predicted and ground truth bounding boxes.
- **Mean Average Precision (mAP)**: Combines precision and recall across all classes and IoU thresholds.
- **Frames Per Second (FPS)**: Measures detection speed, crucial for real-time applications on edge devices.

Pre-Trained Object Detection Models Overview

As we saw in the introduction, given an image or a video stream, an object detection model can identify which of a known set of objects might be present and provide information about their positions within the image.

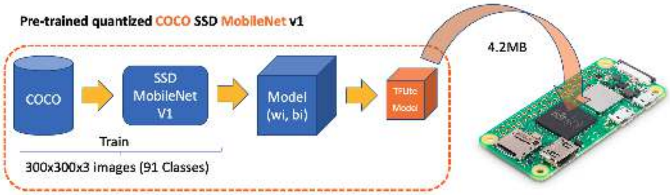
> You can test some common models online by visiting Object Detection - MediaPipe Studio

On Kaggle, we can find the most common pre-trained tflite models to use with the Raspi, ssd_mobilenet_v1, and EfficientDet. Those models were trained on the COCO (Common Objects in Context) dataset, with over 200,000 labeled images in 91 categories. Go, download the models, and upload them to the `./models` folder in the Raspi.

> Alternatively, you can find the models and the COCO labels on GitHub.

For the first part of this lab, we will focus on a pre-trained 300 × 300 SSD-Mobilenet V1 model and compare it with the 320 × 320 EfficientDet-lite0, also trained using the COCO 2017 dataset. Both models were converted to a TensorFlow Lite format (4.2 MB for the SSD Mobilenet and 4.6 MB for the EfficientDet).

> SSD-Mobilenet V2 or V3 is recommended for transfer learning projects, but once the V1 TFLite model is publicly available, we will use it for this overview.


Pre-trained quantized COCO SSD MobileNet v1
COCO
SSD MobileNet V1
Model (wi, bi)
TFLite Model
Train
300x300x3 images (91 Classes)
4.2MB

Setting Up the TFLite Environment

We should confirm the steps done on the last Hands-On Lab, Image Classification, as follows:

- Updating the Raspberry Pi
- Installing Required Libraries
- Setting up a Virtual Environment (Optional but Recommended)

```
source ~/tflite/bin/activate
```

- Installing TensorFlow Lite Runtime
- Installing Additional Python Libraries (inside the environment)

Creating a Working Directory:

Considering that we have created the `Documents/TFLITE` folder in the last Lab, let's now create the specific folders for this object detection lab:

```
cd Documents/TFLITE/
mkdir OBJ_DETECT
cd OBJ_DETECT
mkdir images
mkdir models
cd models
```

Inference and Post-Processing

Let's start a new notebook to follow all the steps to detect objects on an image:

Import the needed libraries:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tflite_runtime.interpreter as tflite
```

Load the TFLite model and allocate tensors:

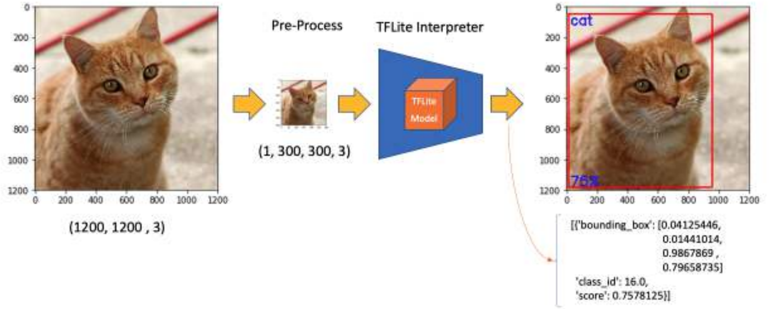
```
model_path = "./models/ssd-mobilenet-v1-tflite-default-v1.tflite"
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
```

Get input and output tensors.

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

Input details will inform us how the model should be fed with an image. The shape of `(1, 300, 300, 3)` with a dtype of `uint8` tells us that a non-normalized (pixel value range from 0 to 255) image with dimensions $(300 \times 300 \times 3)$ should be input one by one (Batch Dimension: 1).

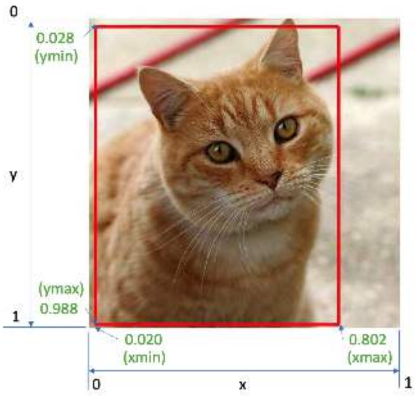
The **output details** include not only the labels ("classes") and probabilities ("scores") but also the relative window position of the bounding boxes ("boxes") about where the object is located on the image and the number of detected objects ("num_detections"). The output details also tell us that the model can detect a **maximum of 10 objects** in the image.



```
[{'bounding_box': [0.04125446,
                   0.01441014,
                   0.9867869 ,
                   0.79658735]
 'class_id': 16.0,
 'score': 0.7578125}]
```

So, for the above example, using the same cat image used with the *Image Classification Lab* looking for the output, we have a **76% probability** of having found an object with a **class ID of 16** on an area delimited by a **bounding box of [0.028011084, 0.020121813, 0.9886069, 0.802299]**. Those four numbers are related to `ymin`, `xmin`, `ymax` and `xmax`, the box coordinates.

Taking into consideration that **y** goes from the top (`ymin`) to the bottom (`ymax`) and **x** goes from left (`xmin`) to the right (`xmax`), we have, in fact, the coordinates of the top/left corner and the bottom/right one. With both edges and knowing the shape of the picture, it is possible to draw a rectangle around the object, as shown in the figure below:

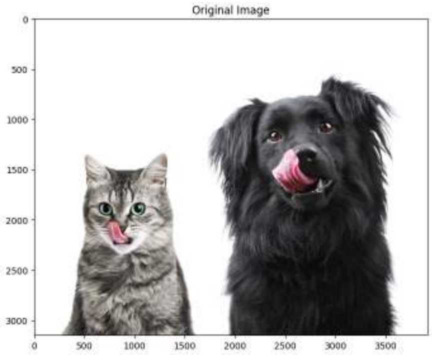


Next, we should find what class ID equal to 16 means. Opening the file `coco_labels.txt`, as a list, each element has an associated index, and inspecting index 16, we get, as expected, `cat`. The probability is the value returning from the score.

Let's now upload some images with multiple objects on it for testing.

```
img_path = "./images/cat_dog.jpeg"
orig_img = Image.open(img_path)
```

```
# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(orig_img)
plt.title("Original Image")
plt.show()
```



Based on the input details, let's pre-process the image, changing its shape and expanding its dimension:

```
img = orig_img.resize((input_details[0]['shape'][1],
                  input_details[0]['shape'][2]))
input_data = np.expand_dims(img, axis=0)
input_data.shape, input_data.dtype
```

The new input_data shape is `(1, 300, 300, 3)` with a dtype of `uint8`, which is compatible with what the model expects.

Using the input_data, let's run the interpreter, measure the latency, and get the output:

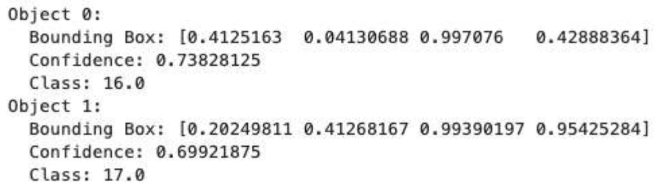
```
start_time = time.time()
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
end_time = time.time()
inference_time = (end_time -
                  start_time) * 1000  # Convert to milliseconds
print ("Inference time: {:.1f}ms".format(inference_time))
```

With a latency of around 800 ms, we can get 4 distinct outputs:

```
boxes = interpreter.get_tensor(output_details[0]['index'])[0]
classes = interpreter.get_tensor(output_details[1]['index'])[0]
scores = interpreter.get_tensor(output_details[2]['index'])[0]
num_detections = int(interpreter.get_tensor(output_details[3]
                     ['index'])[0])
```

On a quick inspection, we can see that the model detected 2 objects with a score over 0.5:

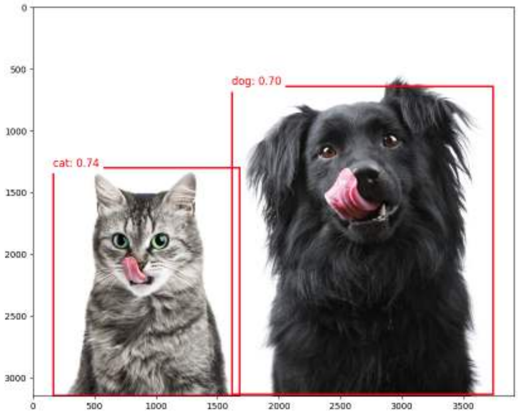
```
for i in range(num_detections):
    if scores[i] > 0.5:  # Confidence threshold
        print(f"Object {i}:")
        print(f"  Bounding Box: {boxes[i]}")
        print(f"  Confidence: {scores[i]}")
        print(f"  Class: {classes[i]}")
```



```
Object 0:
  Bounding Box: [0.4125163  0.04130688 0.997076   0.42888364]
  Confidence: 0.73828125
  Class: 16.0
Object 1:
  Bounding Box: [0.20249811 0.41268167 0.99390197 0.95425284]
  Confidence: 0.69921875
  Class: 17.0
```

And we can also visualize the results:

```
plt.figure(figsize=(12, 8))
plt.imshow(orig_img)
for i in range(num_detections):
   if scores[i] > 0.5:  # Adjust threshold as needed
      ymin, xmin, ymax, xmax = boxes[i]
      (left, right, top, bottom) = (xmin * orig_img.width,
                                    xmax * orig_img.width,
                                    ymin * orig_img.height,
                                    ymax * orig_img.height)
      rect = plt.Rectangle((left, top), right-left, bottom-top,
                           fill=False, color='red', linewidth=2)
      plt.gca().add_patch(rect)
      class_id = int(classes[i])
      class_name = labels[class_id]
      plt.text(left, top-10, f'{class_name}: {scores[i]:.2f}',
               color='red', fontsize=12, backgroundcolor='white')
```



EfficientDet

EfficientDet is not technically an SSD (Single Shot Detector) model, but it shares some similarities and builds upon ideas from SSD and other object detection architectures:

1. EfficientDet:
   - Developed by Google researchers in 2019
   - Uses EfficientNet as the backbone network
   - Employs a novel bi-directional feature pyramid network (BiFPN)
   - It uses compound scaling to scale the backbone network and the object detection components efficiently.
2. Similarities to SSD:
   - Both are single-stage detectors, meaning they perform object localization and classification in a single forward pass.
   - Both use multi-scale feature maps to detect objects at different scales.
3. Key differences:
   - Backbone: SSD typically uses VGG or MobileNet, while EfficientDet uses EfficientNet.
   - Feature fusion: SSD uses a simple feature pyramid, while EfficientDet uses the more advanced BiFPN.
   - Scaling method: EfficientDet introduces compound scaling for all components of the network
4. Advantages of EfficientDet:
   - Generally achieves better accuracy-efficiency trade-offs than SSD and many other object detection models.
   - More flexible scaling allows for a family of models with different size-performance trade-offs.

While EfficientDet is not an SSD model, it can be seen as an evolution of single-stage detection architectures, incorporating more advanced techniques to improve efficiency and accuracy. When using EfficientDet, we can expect similar output structures to SSD (e.g., bounding boxes and class scores).

> On GitHub, you can find another notebook exploring the EfficientDet model that we did with SSD MobileNet.

Object Detection Project

Now, we will develop a complete Image Classification project from data collection to training and deployment. As we did with the Image Classification project, the trained and converted model will be used for inference.

We will use the same dataset to train 3 models: SSD-MobileNet V2, FOMO, and YOLO.

The Goal

All Machine Learning projects need to start with a goal. Let's assume we are in an industrial facility and must sort and count **wheels** and special **boxes**.

In other words, we should perform a multi-label classification, where each image can have three classes:

- Background (no objects)
- Box
- Wheel

Raw Data Collection

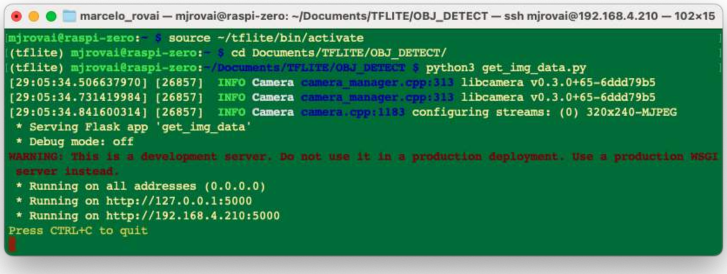
Once we have defined our Machine Learning project goal, the next and most crucial step is collecting the dataset. We can use a phone, the Raspi, or a mix to create the raw dataset (with no labels). Let's use the simple web app on our Raspberry Pi to view the `QVGA (320 x 240)` captured images in a browser.

From GitHub, get the Python script get_img_data.py and open it in the terminal:

```
python3 get_img_data.py
```

Access the web interface:

- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`
- From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace`with your Raspberry Pi's IP address). For example:`http://192.168.4.210:5000/‘

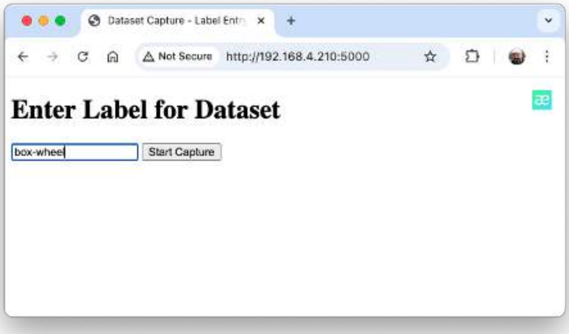


marcelo_rovai — mjrovai@raspi-zero: ~/Documents/TFLITE/OBJ_DETECT — ssh mjrovai@192.168.4.210 — 102×15

```
mjrovai@raspi-zero:~ $ source ~/tflite/bin/activate
(tflite) mjrovai@raspi-zero:~ $ cd Documents/TFLITE/OBJ_DETECT/
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT $ python3 get_img_data.py
[29:05:34.506637970] [26857]  INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5
[29:05:34.731419984] [26857]  INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5
[29:05:34.841600314] [26857]  INFO Camera camera.cpp:1183 configuring streams: (0) 320x240-MJPEG
 * Serving Flask app 'get_img_data'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
 server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.4.210:5000
Press CTRL+C to quit
```

The Python script creates a web-based interface for capturing and organizing image datasets using a Raspberry Pi and its camera. It's handy for machine learning projects that require labeled image data or not, as in our case here.

Access the web interface from a browser, enter a generic label for the images you want to capture, and press `Start Capture`.

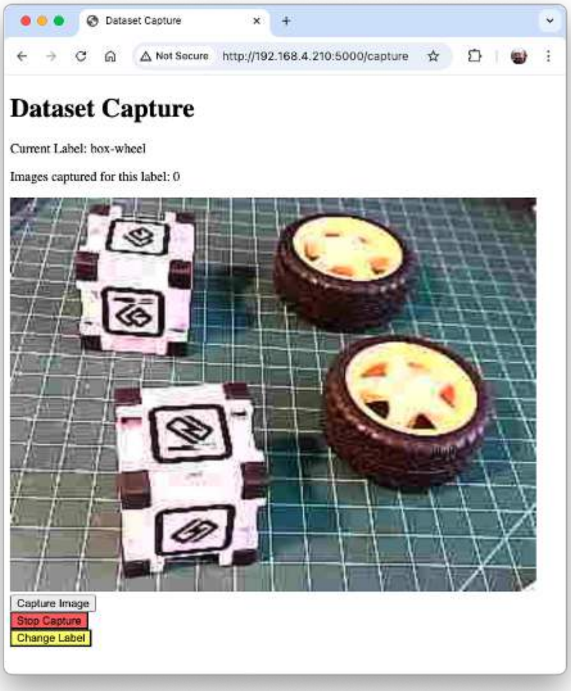


Enter Label for Dataset

box-wheel

> Note that the images to be captured will have multiple labels that should be defined later.

Use the live preview to position the camera and click `Capture Image` to save images under the current label (in this case, `box-wheel`.

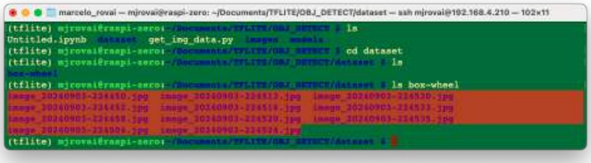


Dataset Capture

Current Label: box-wheel

Images captured for this label: 0

When we have enough images, we can press `Stop Capture`. The captured images are saved on the folder dataset/box-wheel:



```
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/TFLITE/OBJ_DETECT/dataset — ssh mjrovai@192.168.4.210 — 102×11
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT $ ls
Untitled.ipynb  dataset  get_img_data.py  images  models
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT $ cd dataset
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT/dataset $ ls
box-wheel
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT/dataset $ ls box-wheel
image_20240903-224450.jpg  image_20240903-224512.jpg  image_20240903-224530.jpg
image_20240903-224452.jpg  image_20240903-224514.jpg  image_20240903-224533.jpg
image_20240903-224458.jpg  image_20240903-224520.jpg  image_20240903-224535.jpg
image_20240903-224506.jpg  image_20240903-224526.jpg
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT/dataset $ 
```

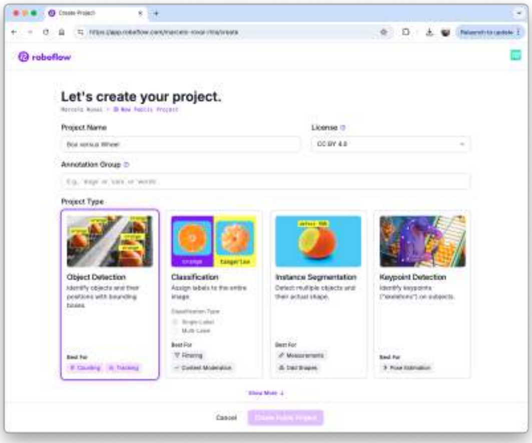
> Get around 60 images. Try to capture different angles, backgrounds, and light conditions. Filezilla can transfer the created raw dataset to your main computer.

Labeling Data

The next step in an Object Detect project is to create a labeled dataset. We should label the raw dataset images, creating bounding boxes around each picture's objects (box and wheel). We can use labeling tools like LabelImg, CVAT, Roboflow, or even the Edge Impulse Studio. Once we have explored the Edge Impulse tool in other labs, let's use Roboflow here.

> We are using Roboflow (free version) here for two main reasons. 1) We can have auto-labeler, and 2) The annotated dataset is available in several formats and can be used both on Edge Impulse Studio (we will use it for MobileNet V2 and FOMO train) and on CoLab (YOLOv8 train), for example. Having the annotated dataset on Edge Impulse (Free account), it is not possible to use it for training on other platforms.

We should upload the raw dataset to Roboflow. Create a free account there and start a new project, for example, ("box-versus-wheel").



Let's create your project.

Project Name: Box versus Wheel

License: CC BY 4.0

Annotation Group: E.g. 'dogs' or 'cars' or 'words'

Project Type

Object Detection: Identify objects and their positions with bounding boxes. Best For: Counting, Tracking

Classification: Assign labels to the entire image. Classification Type: Single-Label, Multi-Label. Best For: Filtering, Content Moderation

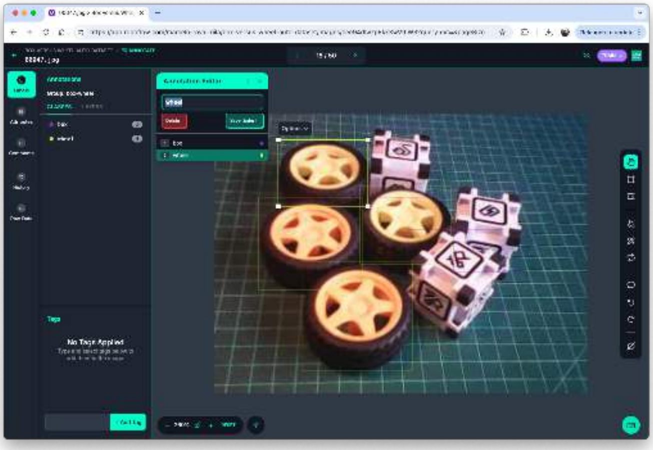
Instance Segmentation: Detect multiple objects and their actual shape. Best For: Measurements, Odd Shapes

Keypoint Detection: Identify keypoints ("skeletons") on subjects. Best For: Pose Estimation

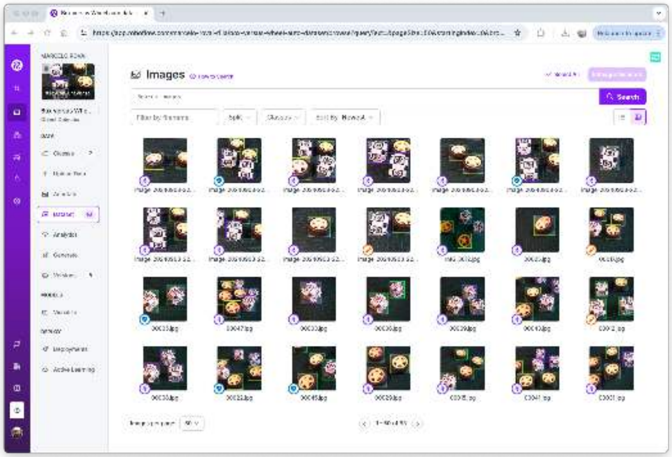
> We will not enter in deep details about the Roboflow process once many tutorials are available.

Annotate

Once the project is created and the dataset is uploaded, you should make the annotations using the "Auto-Label" Tool. Note that you can also upload images with only a background, which should be saved w/o any annotations.



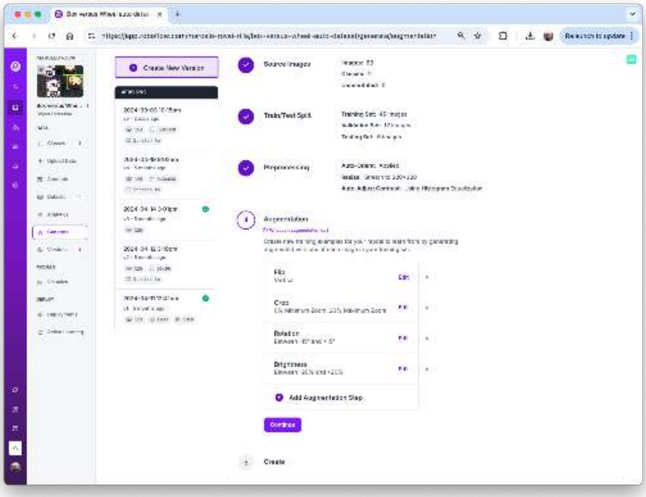
Once all images are annotated, you should split them into training, validation, and testing.



Data Pre-Processing

The last step with the dataset is preprocessing to generate a final version for training. Let's resize all images to 320 × 320 and generate augmented versions of each image (augmentation) to create new training examples from which our model can learn.

For augmentation, we will rotate the images (+/-15º), crop, and vary the brightness and exposure.



Source Images

Train/Test Split

Training Set: 45 images

Preprocessing

Auto-Orient: Applied

Resize: Stretch to 320×320

Auto-Adjust Contrast: Using Histogram Equalization

Augmentation

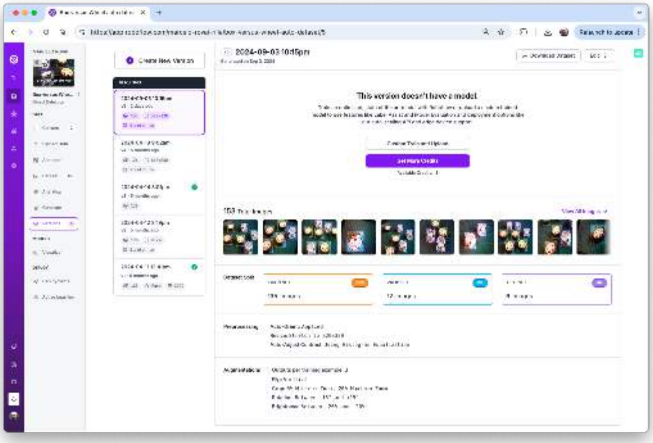
Flip
Vertical

Crop
0% Minimum Zoom, 20% Maximum Zoom

Rotation
Between -15° and +15°

Brightness
Between -20% and +20%

At the end of the process, we will have 153 images.

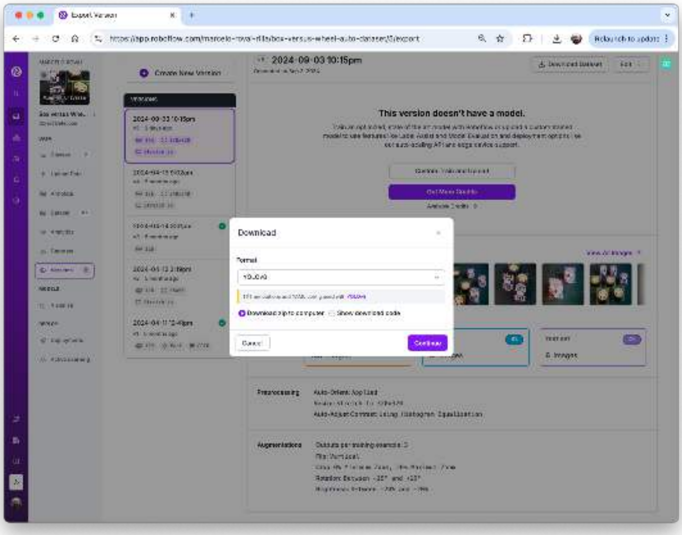


2024-09-03 10:15pm

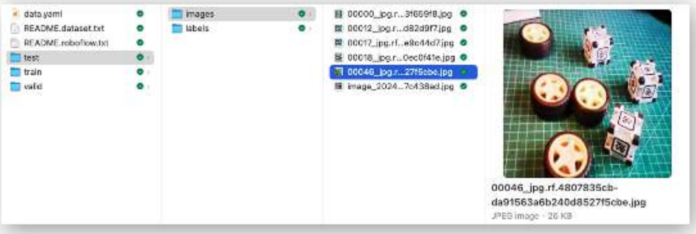
This version doesn't have a model.

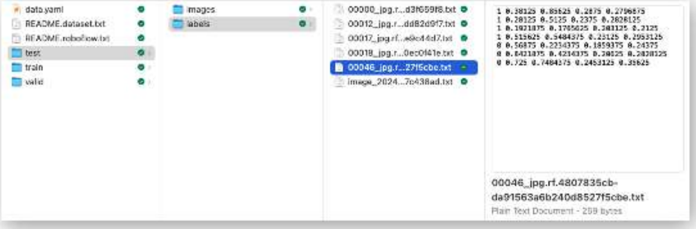
153 Total Images

Now, you should export the annotated dataset in a format that Edge Impulse, Ultralitics, and other frameworks/tools understand, for example, `YOLOv8`. Let's download a zipped version of the dataset to our desktop.



Here, it is possible to review how the dataset was structured


data.yaml
README.dataset.txt
README.roboflow.txt
test
train
valid
images
labels
00000_jpg.r...3f659f8.jpg
00012_jpg.r...d82d9f7.jpg
00017_jpg.rf...e8c44d7.jpg
00018_jpg.r...0ec0f41e.jpg
00046_jpg.r...27f5cbe.jpg
image_2024...7c438ad.jpg
00046_jpg.rf.4807835cb-da91563a6b240d85271f5cbe.jpg
JPEG image - 26 KB


data.yaml
README.dataset.txt
README.roboflow.txt
test
train
valid
images
labels
00000_jpg.r...d3f659f8.txt
00012_jpg.r...dd82d9f7.txt
00017_jpg.rf...e8c44d7.txt
00018_jpg.r...0ec0f41e.txt
00046_jpg.r...27f5cbe.txt
image_2024...7c438ad.txt
00046_jpg.rf.4807835cb-da91563a6b240d85271f5cbe.txt
Plain Text Document - 259 bytes

```
1 0.38125 0.85625 0.2875 0.2796875
1 0.28125 0.5125 0.2375 0.2828125
1 0.1921875 0.1765625 0.203125 0.2125
1 0.515625 0.5484375 0.23125 0.2953125
0 0.56875 0.2234375 0.1859375 0.24375
0 0.8421875 0.4234375 0.20625 0.2828125
0 0.725 0.7484375 0.2453125 0.35625
```

There are 3 separate folders, one for each split (`train/test/valid`). For each of them, there are 2 subfolders, `images`, and `labels`. The pictures are stored as **image_id.jpg** and **images_id.txt**, where "image_id" is unique for every picture.

The labels file format will be `class_id bounding box coordinates`, where in our case, class_id will be `0` for `box` and `1` for `wheel`. The numerical id (o, 1, 2...) will follow the alphabetical order of the class name.

The `data.yaml` file has info about the dataset as the classes' names (`names: ['box', 'wheel']`) following the YOLO format.

And that's it! We are ready to start training using the Edge Impulse Studio (as we will do in the following step), Ultralytics (as we will when discussing YOLO), or even training from scratch on CoLab (as we did with the Cifar-10 dataset on the Image Classification lab).

> The pre-processed dataset can be found at the Roboflow site, or here:

Training an SSD MobileNet Model on Edge Impulse Studio

Go to Edge Impulse Studio, enter your credentials at **Login** (or create an account), and start a new project.

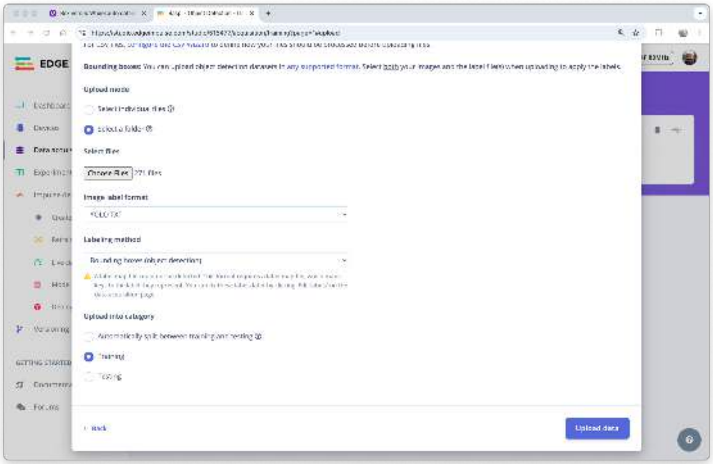
> Here, you can clone the project developed for this hands-on lab: Raspi - Object Detection.

On the Project `Dashboard` tab, go down and on **Project info,** and for Labeling method select `Bounding boxes (object detection)`

Uploading the annotated data

On Studio, go to the `Data acquisition` tab, and on the `UPLOAD DATA` section, upload from your computer the raw dataset.

We can use the option `Select a folder`, choosing, for example, the folder `train` in your computer, which contains two sub-folders, `images`, and `labels`. Select the `Image label format`, "YOLO TXT", upload into the caegory `Training`, and press `Upload data`.



Bounding boxes: You can upload object detection datasets in any supported format. Select both your images and the label files when uploading to apply the labels.

Upload mode

○ Select individual files

● Select a folder

Select files

Choose files 271 files

Image label format

YOLO TXT

Labeling method

Bounding boxes (object detection)

Upload into category

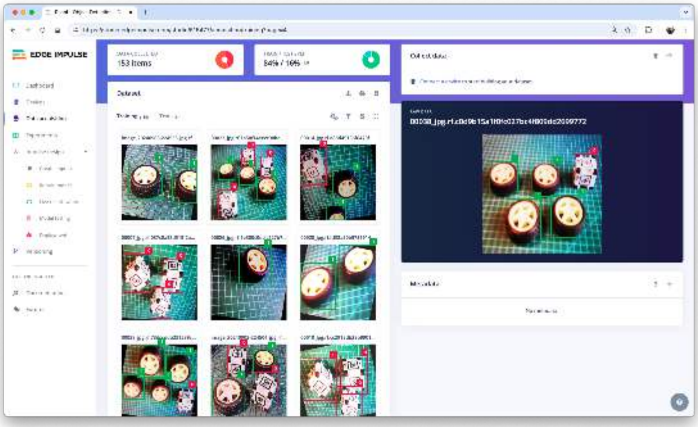
○ Automatically split between training and testing

● Training

○ Testing

Repeat the process for the test data (upload both folders, test, and validation). At the end of the upload process, you should end with the annotated dataset of 153 images split in the train/test (84%/16%).

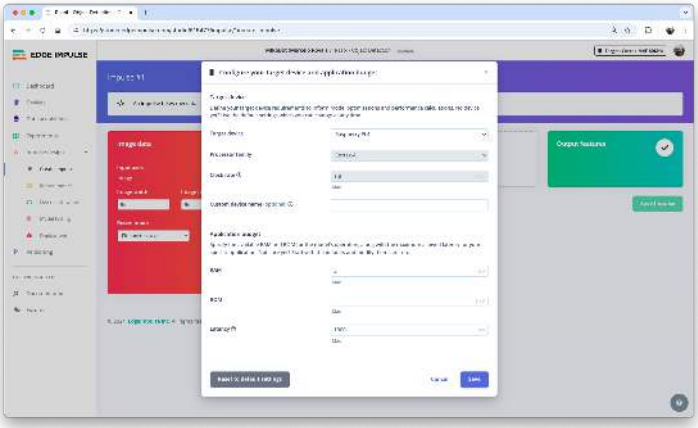
> Note that labels will be stored at the labels files `0` and `1` , which are equivalent to `box` and `wheel`.



The Impulse Design

The first thing to define when we enter the `Create impulse` step is to describe the target device for deployment. A pop-up window will appear. We will select Raspberry 4, an intermediary device between the Raspi-Zero and the Raspi-5.

> This choice will not interfere with the training; it will only give us an idea about the latency of the model on that specific target.



In this phase, you should define how to:

- **Pre-processing** consists of resizing the individual images. In our case, the images were pre-processed on Roboflow, to `320x320` , so let's keep it. The resize will not matter here because the images are already squared. If you upload a rectangular image, squash it (squared form, without cropping). Afterward, you could define if the images are converted from RGB to Grayscale or not.
- **Design a Model,** in this case, "Object Detection."

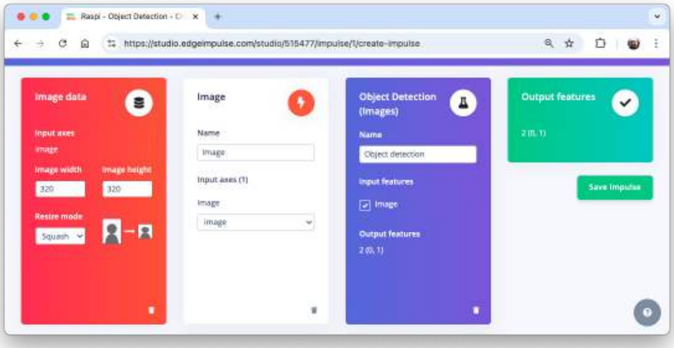

https://studio.edgeimpulse.com/studio/515477/impulse/1/create-impulse

Image data
Input axes: image
Image width: 320
Image height: 320
Resize mode: Squash

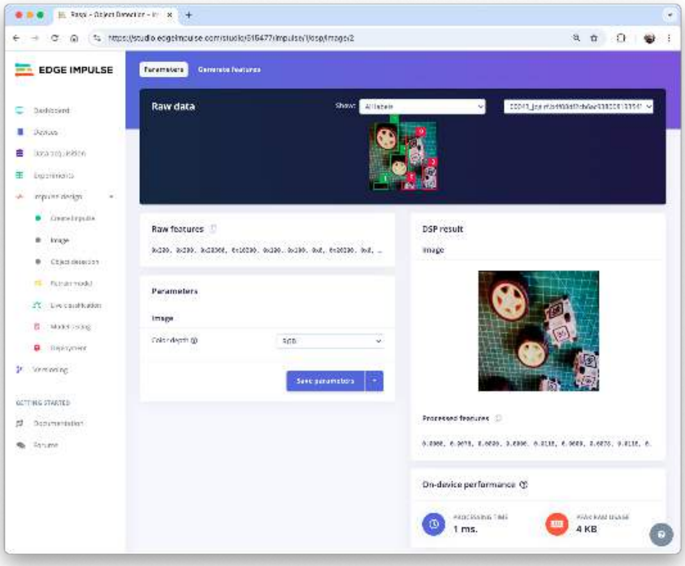
Image
Name: Image
Input axes (1): image

Object Detection (Images)
Name: Object detection
Input features: ☑ Image
Output features: 2 (0, 1)

Output features
2 (0, 1)

Preprocessing all dataset

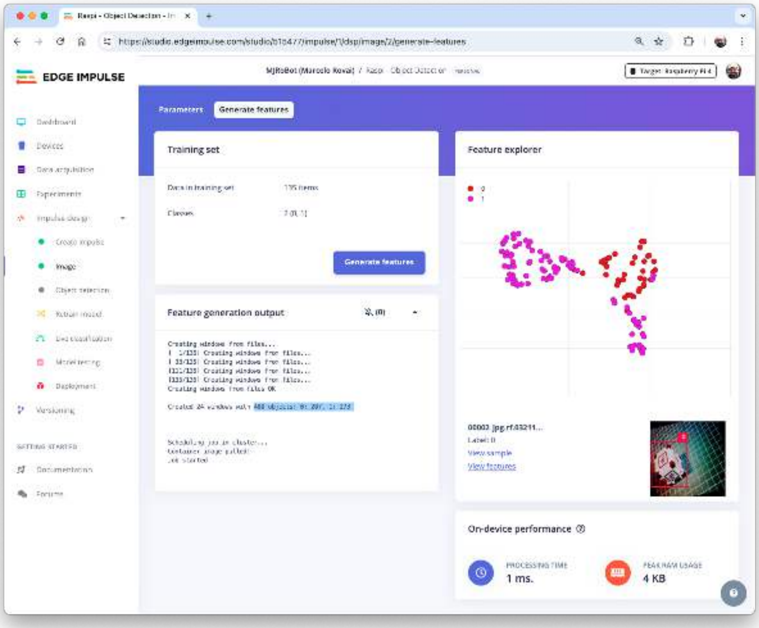
In the section `Image`, select **Color depth** as `RGB`, and press `Save parameters`.


Parameters
Image
Color depth: RGB

DSP result
Image

On-device performance
Processing time: 1 ms.
Peak RAM usage: 4 KB

The Studio moves automatically to the next section, `Generate features`, where all samples will be pre-processed, resulting in 480 objects: 207 boxes and 273 wheels.



Training set

Data in training set: 135 items

Classes: 2 (0, 1)

Generate features

Feature generation output

```
Creating windows from files...
[  1/135] Creating windows from files...
[ 33/135] Creating windows from files...
[133/135] Creating windows from files...
[135/135] Creating windows from files...
Creating windows from files OK

Created 24 windows with 480 objects: 0: 207, 1: 273


Scheduling job in cluster...
Container image pulled!
Job started
```

00003.jpg.rf.43211...
Label: 0
View sample
View features

On-device performance

PROCESSING TIME: 1 ms.

PEAK RAM USAGE: 4 KB

The feature explorer shows that all samples evidence a good separation after the feature generation.

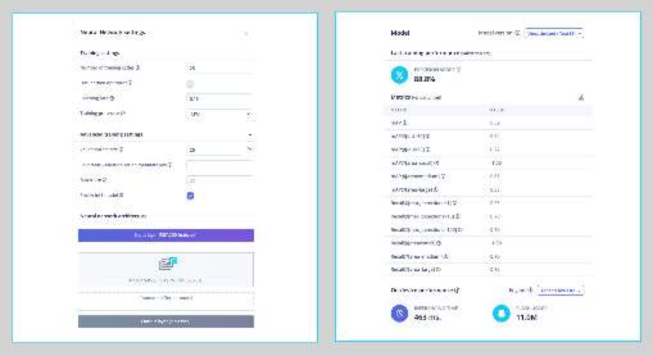
Model Design, Training, and Test

For training, we should select a pre-trained model. Let's use the **MobileNetV2 SSD FPN-Lite (320x320 only)** . It is a pre-trained object detection model designed to locate up to 10 objects within an image, outputting a bounding box for each object detected. The model is around 3.7 MB in size. It supports an RGB input at 320 × 320 px.

Regarding the training hyper-parameters, the model will be trained with:

- Epochs: 25
- Batch size: 32
- Learning Rate: 0.15.

For validation during training, 20% of the dataset (*validation_dataset*) will be spared.



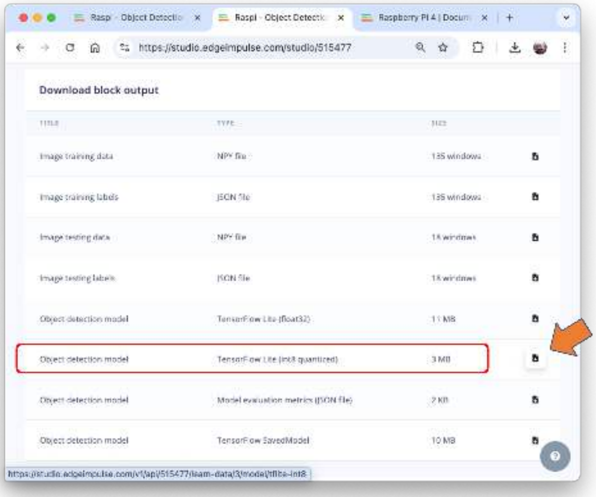
As a result, the model ends with an overall precision score (based on COCO mAP) of 88.8%, higher than the result when using the test data (83.3%).

Deploying the model

We have two ways to deploy our model:

- **TFLite model**, which lets deploy the trained model as `.tflite` for the Raspi to run it using Python.
- **Linux (AARCH64)**, a binary for Linux (AARCH64), implements the Edge Impulse Linux protocol, which lets us run our models on any Linux-based development board, with SDKs for Python, for example. See the documentation for more information and setup instructions.

Let's deploy the **TFLite model**. On the `Dashboard` tab, go to Transfer learning model (int8 quantized) and click on the download icon:



Download block output

TITLE	TYPE	SIZE
Image training data	NPY file	135 windows
Image training labels	JSON file	135 windows
Image testing data	NPY file	18 windows
Image testing labels	JSON file	18 windows
Object detection model	TensorFlow Lite (float32)	11 MB
Object detection model	TensorFlow Lite (int8 quantized)	3 MB
Object detection model	Model evaluation metrics (JSON file)	2 KB
Object detection model	TensorFlow SavedModel	10 MB

https://studio.edgeimpulse.com/v1/api/515477/learn-data/3/model/tflite-int8

Transfer the model from your computer to the Raspi folder `./models` and capture or get some images for inference and save them in the folder `./images`.

Inference and Post-Processing

The inference can be made as discussed in the *Pre-Trained Object Detection Models Overview*. Let's start a new notebook to follow all the steps to detect cubes and wheels on an image.

Import the needed libraries:

```
import time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image
import tflite_runtime.interpreter as tflite
```

Define the model path and labels:

```
model_path = "./models/ei-raspi-object-detection-SSD-\
              MobileNetv2-320x0320-int8.lite"
labels = ['box', 'wheel']
```

> Remember that the model will output the class ID as values (0 and 1), following an alphabetic order regarding the class names.

Load the model, allocate the tensors, and get the input and output tensor details:

```
# Load the TFLite model
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()

# Get input and output tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

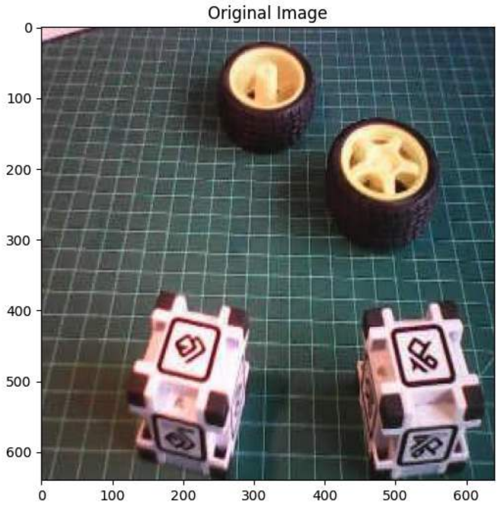
One crucial difference to note is that the `dtype` of the input details of the model is now `int8`, which means that the input values go from –128 to +127, while each pixel of our raw image goes from 0 to 256. This means that we should pre-process the image to match it. We can check here:

```
input_dtype = input_details[0]['dtype']
input_dtype
```

```
numpy.int8
```

So, let's open the image and show it:

```
# Load the image
img_path = "./images/box_2_wheel_2.jpg"
orig_img = Image.open(img_path)

# Display the image
plt.figure(figsize=(6, 6))
plt.imshow(orig_img)
plt.title("Original Image")
plt.show()
```



And perform the pre-processing:

```
scale, zero_point = input_details[0]['quantization']
img = orig_img.resize((input_details[0]['shape'][1],
                       input_details[0]['shape'][2]))
img_array = np.array(img, dtype=np.float32) / 255.0
img_array = (
    (img_array / scale + zero_point)
    .clip(-128, 127)
    .astype(np.int8)
)
input_data = np.expand_dims(img_array, axis=0)
```

Checking the input data, we can verify that the input tensor is compatible with what is expected by the model:

```
input_data.shape, input_data.dtype
```

```
((1, 320, 320, 3), dtype('int8'))
```

Now, it is time to perform the inference. Let's also calculate the latency of the model:

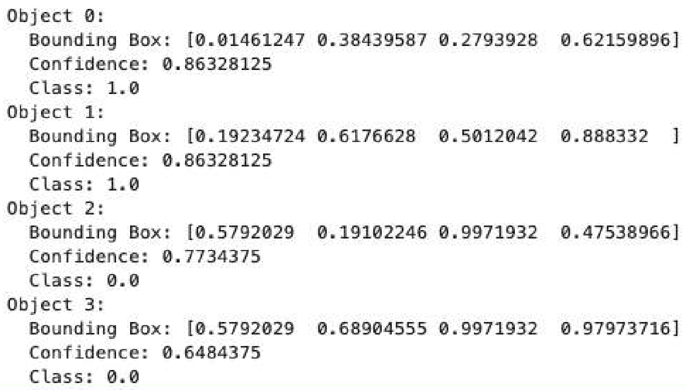
```
# Inference on Raspi-Zero
start_time = time.time()
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
end_time = time.time()
inference_time = (
    (end_time - start_time)
    * 1000  # Convert to milliseconds
)
print ("Inference time: {:.1f}ms".format(inference_time))
```

The model will take around 600ms to perform the inference in the Raspi-Zero, which is around 5 times longer than a Raspi-5.

Now, we can get the output classes of objects detected, its bounding boxes coordinates, and probabilities.

```
boxes = interpreter.get_tensor(output_details[1]['index'])[0]
classes = interpreter.get_tensor(output_details[3]['index'])[0]
scores = interpreter.get_tensor(output_details[0]['index'])[0]
num_detections = int(
    interpreter.get_tensor(
        output_details[2]['index']
    )[0]
)
```

```
for i in range(num_detections):
    if scores[i] > 0.5:  # Confidence threshold
        print(f"Object {i}:")
        print(f"  Bounding Box: {boxes[i]}")
        print(f"  Confidence: {scores[i]}")
        print(f"  Class: {classes[i]}")
```

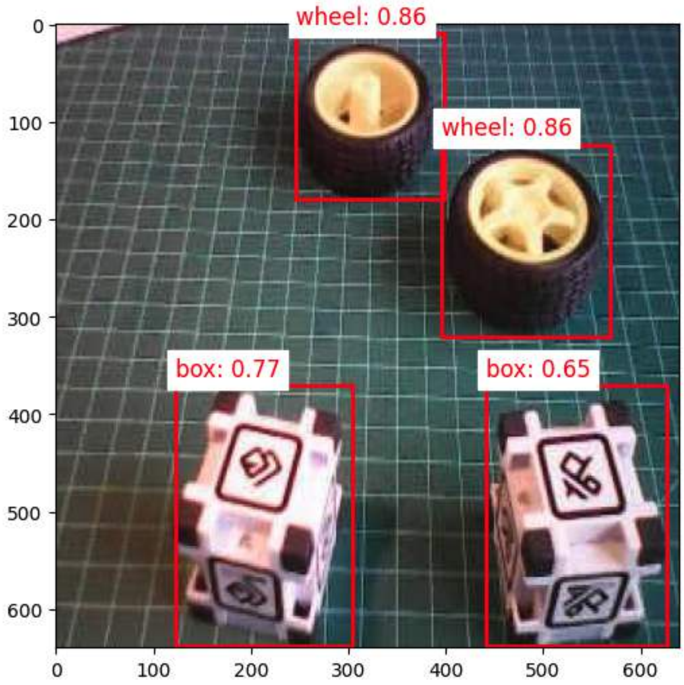


```
Object 0:
  Bounding Box: [0.01461247 0.38439587 0.2793928  0.62159896]
  Confidence: 0.86328125
  Class: 1.0
Object 1:
  Bounding Box: [0.19234724 0.6176628  0.5012042  0.888332  ]
  Confidence: 0.86328125
  Class: 1.0
Object 2:
  Bounding Box: [0.5792029  0.19102246 0.9971932  0.47538966]
  Confidence: 0.7734375
  Class: 0.0
Object 3:
  Bounding Box: [0.5792029  0.68904555 0.9971932  0.97973716]
  Confidence: 0.6484375
  Class: 0.0
```

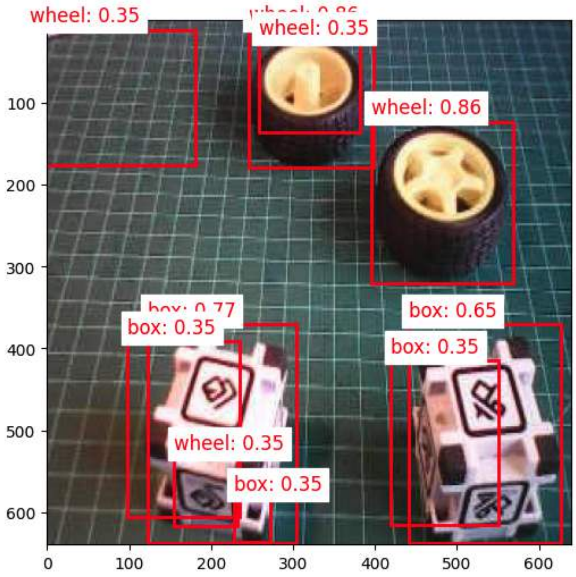
From the results, we can see that 4 objects were detected: two with class ID 0 (`box`)and two with class ID 1 (`wheel`), what is correct!

Let's visualize the result for a `threshold` of 0.5

```
threshold = 0.5
plt.figure(figsize=(6,6))
plt.imshow(orig_img)
for i in range(num_detections):
   if scores[i] > threshold:
      ymin, xmin, ymax, xmax = boxes[i]
      (left, right, top, bottom) = (xmin * orig_img.width,
                                    xmax * orig_img.width,
                                    ymin * orig_img.height,
                                    ymax * orig_img.height)
      rect = plt.Rectangle((left, top), right-left, bottom-top,
                           fill=False, color='red', linewidth=2)
      plt.gca().add_patch(rect)
      class_id = int(classes[i])
      class_name = labels[class_id]
      plt.text(left, top-10, f'{class_name}: {scores[i]:.2f}',
               color='red', fontsize=12, backgroundcolor='white')
```



But what happens if we reduce the threshold to 0.3, for example?



We start to see false positives and **multiple detections**, where the model detects the same object multiple times with different confidence levels and slightly different bounding boxes.

Commonly, sometimes, we need to adjust the threshold to smaller values to capture all objects, avoiding false negatives, which would lead to multiple detections.

To improve the detection results, we should implement **Non-Maximum Suppression (NMS)**, which helps eliminate overlapping bounding boxes and keeps only the most confident detection.

For that, let's create a general function named `non_max_suppression()`, with the role of refining object detection results by eliminating redundant and overlapping bounding boxes. It achieves this by iteratively selecting the detection with the highest confidence score and removing other significantly overlapping detections based on an Intersection over Union (IoU) threshold.

```
def non_max_suppression(boxes, scores, threshold):
    # Convert to corner coordinates
    x1 = boxes[:, 0]
    y1 = boxes[:, 1]
    x2 = boxes[:, 2]
    y2 = boxes[:, 3]

    areas = (x2 - x1 + 1) * (y2 - y1 + 1)
    order = scores.argsort()[::-1]

    keep = []
    while order.size > 0:
        i = order[0]
        keep.append(i)
        xx1 = np.maximum(x1[i], x1[order[1:]])
        yy1 = np.maximum(y1[i], y1[order[1:]])
        xx2 = np.minimum(x2[i], x2[order[1:]])
        yy2 = np.minimum(y2[i], y2[order[1:]])

        w = np.maximum(0.0, xx2 - xx1 + 1)
        h = np.maximum(0.0, yy2 - yy1 + 1)
        inter = w * h
        ovr = inter / (areas[i] + areas[order[1:]] - inter)

        inds = np.where(ovr <= threshold)[0]
        order = order[inds + 1]

    return keep
```

How it works:

1. Sorting: It starts by sorting all detections by their confidence scores, highest to lowest.

2. Selection: It selects the highest-scoring box and adds it to the final list of detections.
3. Comparison: This selected box is compared with all remaining lower-scoring boxes.
4. Elimination: Any box that overlaps significantly (above the IoU threshold) with the selected box is eliminated.
5. Iteration: This process repeats with the next highest-scoring box until all boxes are processed.

Now, we can define a more precise visualization function that will take into consideration an IoU threshold, detecting only the objects that were selected by the `non_max_suppression` function:

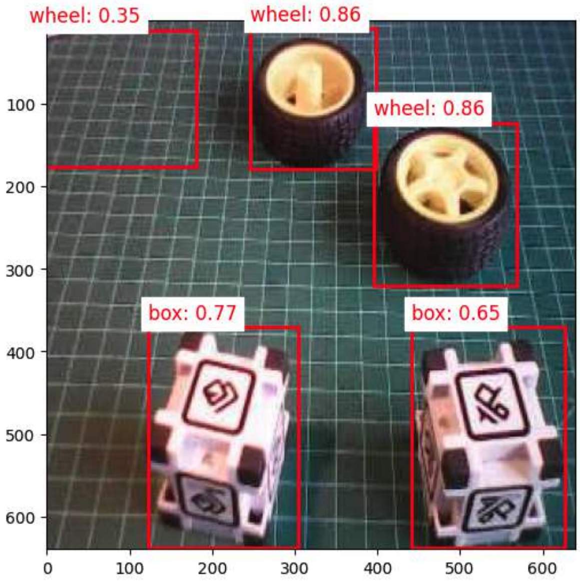
```
def visualize_detections(image, boxes, classes, scores,
                         labels, threshold, iou_threshold):
    if isinstance(image, Image.Image):
        image_np = np.array(image)
    else:
        image_np = image
    height, width = image_np.shape[:2]
    # Convert normalized coordinates to pixel coordinates
    boxes_pixel = boxes * np.array([height, width, height, width])
    # Apply NMS
    keep = non_max_suppression(boxes_pixel, scores, iou_threshold)
    # Set the figure size to 12x8 inches
    fig, ax = plt.subplots(1, figsize=(12, 8))
    ax.imshow(image_np)
    for i in keep:
        if scores[i] > threshold:
            ymin, xmin, ymax, xmax = boxes[i]
            rect = patches.Rectangle(
                (xmin * width, ymin * height),
                (xmax - xmin) * width,
                (ymax - ymin) * height,
                linewidth=2,
                edgecolor='r',
                facecolor='none'
             )

            ax.add_patch(rect)
            class_name = labels[int(classes[i])]
            ax.text(xmin * width, ymin * height - 10,
                    f'{class_name}: {scores[i]:.2f}', color='red',
                    fontsize=12, backgroundcolor='white')
    plt.show()
```

Now we can create a function that will call the others, performing inference on any image:

```
def detect_objects(img_path, conf=0.5, iou=0.5):
    orig_img = Image.open(img_path)
    scale, zero_point = input_details[0]['quantization']
    img = orig_img.resize((input_details[0]['shape'][1],
                      input_details[0]['shape'][2]))
    img_array = np.array(img, dtype=np.float32) / 255.0
    img_array = (img_array / scale + zero_point).\
    clip(-128, 127).astype(np.int8)
    input_data = np.expand_dims(img_array, axis=0)

    # Inference on Raspi-Zero
    start_time = time.time()
    interpreter.set_tensor(input_details[0]['index'], input_data)
    interpreter.invoke()
    end_time = time.time()
    inference_time = (
       end_time - start_time
    ) * 1000  # Convert to milliseconds

    print ("Inference time: {:.1f}ms".format(inference_time))

    # Extract the outputs
    boxes = interpreter.get_tensor(output_details[1]['index'])[0]
    classes = interpreter.get_tensor(
       output_details[3]['index']
    )[0]
    scores = interpreter.get_tensor(
        output_details[0]['index']
    )[0]
    num_detections = int(
        interpreter.get_tensor(
          output_details[2]['index']
        )[0]
    )

    visualize_detections(orig_img, boxes, classes,
                         scores, labels, threshold=conf,
                         iou_threshold=iou)
```

Now, running the code, having the same image again with a confidence threshold of 0.3, but with a small IoU:

```
img_path = "./images/box_2_wheel_2.jpg"
detect_objects(img_path, conf=0.3,iou=0.05)
```



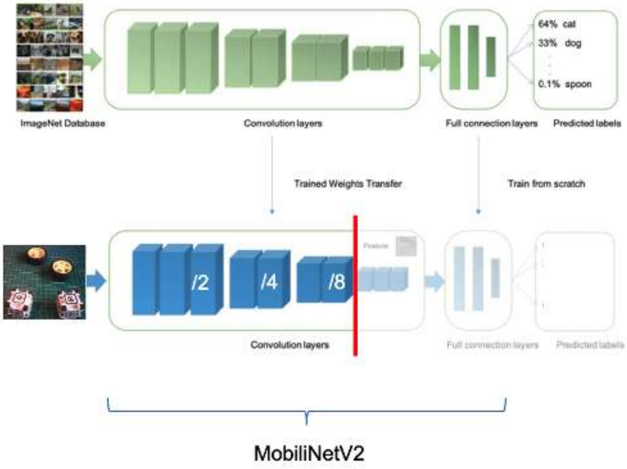
Training a FOMO Model at Edge Impulse Studio

The inference with the SSD MobileNet model worked well, but the latency was significantly high. The inference varied from 0.5 to 1.3 seconds on a Raspi-Zero, which means around or less than 1 FPS (1 frame per second). One alternative to speed up the process is to use FOMO (Faster Objects, More Objects).

This novel machine learning algorithm lets us count multiple objects and find their location in an image in real-time using up to 30× less processing power and memory than MobileNet SSD or YOLO. The main reason this is possible is that while other models calculate the object's size by drawing a square around it (bounding box), FOMO ignores the size of the image, providing only the information about where the object is located in the image through its centroid coordinates.

How FOMO works?

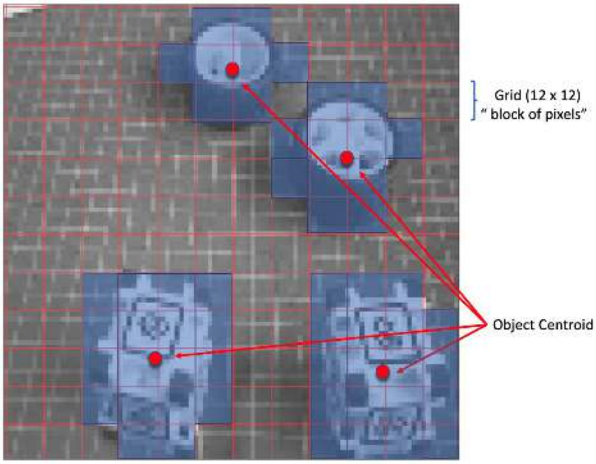
In a typical object detection pipeline, the first stage is extracting features from the input image. **FOMO leverages MobileNetV2 to perform this task**. MobileNetV2 processes the input image to produce a feature map that captures essential characteristics, such as textures, shapes, and object edges, in a computationally efficient way.



Once these features are extracted, FOMO's simpler architecture, focused on center-point detection, interprets the feature map to determine where objects are located in the image. The output is a grid of cells, where each cell represents whether or not an object center is detected. The model outputs one or more confidence scores for each cell, indicating the likelihood of an object being present.

Let's see how it works on an image.

FOMO divides the image into blocks of pixels using a factor of 8. For the input of 96×96, the grid would be 12×12 $(96/8 = 12)$. For a 160×160, the grid will be 20×20, and so on. Next, FOMO will run a classifier through each pixel block to calculate the probability that there is a box or a wheel in each of them and, subsequently, determine the regions that have the highest probability of containing the object (If a pixel block has no objects, it will be classified as *background*). From the overlap of the final region, the FOMO provides the coordinates (related to the image dimensions) of the centroid of this region.

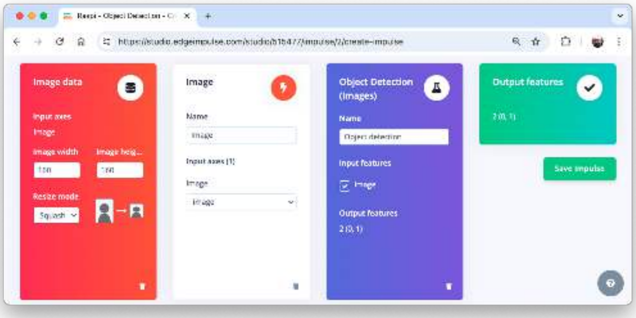


Trade-off Between Speed and Precision:

- **Grid Resolution**: FOMO uses a grid of fixed resolution, meaning each cell can detect if an object is present in that part of the image. While it doesn't provide high localization accuracy, it makes a trade-off by being fast and computationally light, which is crucial for edge devices.
- **Multi-Object Detection**: Since each cell is independent, FOMO can detect multiple objects simultaneously in an image by identifying multiple centers.

Impulse Design, new Training and Testing

Return to Edge Impulse Studio, and in the `Experiments` tab, create another impulse. Now, the input images should be 160 × 160 (this is the expected input size for MobilenetV2).


Raspi - Object Detection - C
https://studio.edgeimpulse.com/studio/515477/impulse/2/create-impulse
Image data
Image
Object Detection (Images)
Output features
Save impulse

Input axes: Image
Image width: 160
Image height: 160
Resize mode: Squash

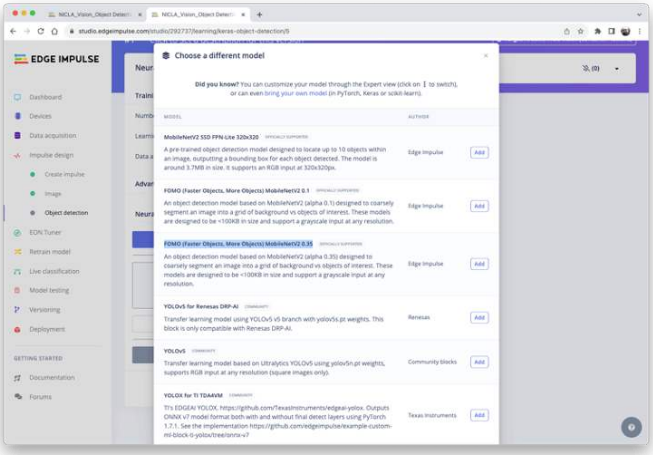
Name: Image
Input axes (1)
Image: Image

Name: Object detection
Input features: ☑ Image
Output features: 2 (0, 1)

Output features: 2 (0, 1)

On the `Image` tab, generate the features and go to the `Object detection` tab.

We should select a pre-trained model for training. Let's use the **FOMO (Faster Objects, More Objects) MobileNetV2 0.35.**



Choose a different model

Did you know? You can customize your model through the Expert view (click on ⋮ to switch), or can even bring your own model (in PyTorch, Keras or scikit-learn).

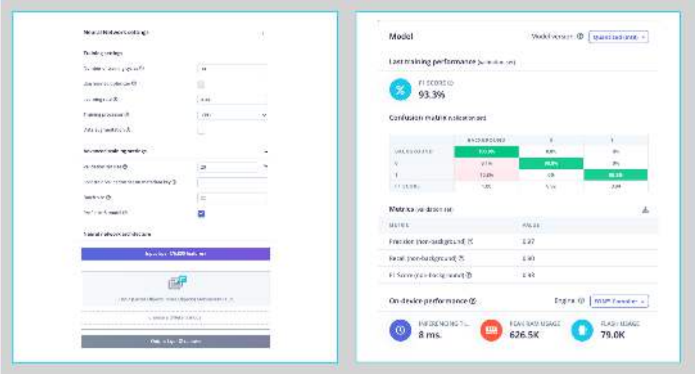
MODEL	AUTHOR	
MobileNetV2 SSD FPN-Lite 320x320 OFFICIALLY SUPPORTED A pre-trained object detection model designed to locate up to 10 objects within an image, outputting a bounding box for each object detected. The model is around 3.7MB in size. It supports an RGB input at 320x320px.	Edge Impulse	Add
FOMO (Faster Objects, More Objects) MobileNetV2 0.1 OFFICIALLY SUPPORTED An object detection model based on MobileNetV2 (alpha 0.1) designed to coarsely segment an image into a grid of background vs objects of interest. These models are designed to be <100KB in size and support a grayscale input at any resolution.	Edge Impulse	Add
FOMO (Faster Objects, More Objects) MobileNetV2 0.35 OFFICIALLY SUPPORTED An object detection model based on MobileNetV2 (alpha 0.35) designed to coarsely segment an image into a grid of background vs objects of interest. These models are designed to be <100KB in size and support a grayscale input at any resolution.	Edge Impulse	Add
YOLOv5 for Renesas DRP-AI COMMUNITY Transfer learning model using YOLOv5 v5 branch with yolov5s.pt weights. This block is only compatible with Renesas DRP-AI.	Renesas	Add
YOLOv5 COMMUNITY Transfer learning model based on Ultralytics YOLOv5 using yolov5n.pt weights, supports RGB input at any resolution (square images only).	Community blocks	Add
YOLOX for TI TDA4VM COMMUNITY TI's EDGEAI YOLOX. https://github.com/TexasInstruments/edgeai-yolox. Outputs ONNX v7 model format both with and without final detect layers using PyTorch 1.7.1. See the implementation https://github.com/edgeimpulse/example-custom-ml-block-ti-yolox/tree/onnx-v7	Texas Instruments	Add

Regarding the training hyper-parameters, the model will be trained with:

- Epochs: 30
- Batch size: 32
- Learning Rate: 0.001.

For validation during training, 20% of the dataset (*validation_dataset*) will be spared. We will not apply Data Augmentation for the remaining 80% (*train_dataset*) because our dataset was already augmented during the labeling phase at Roboflow.

As a result, the model ends with an overall F1 score of 93.3% with an impressive latency of 8 ms (Raspi-4), around 60× less than we got with the SSD MovileNetV2.



Model

Last training performance

F1 SCORE
93.3%

On-device performance

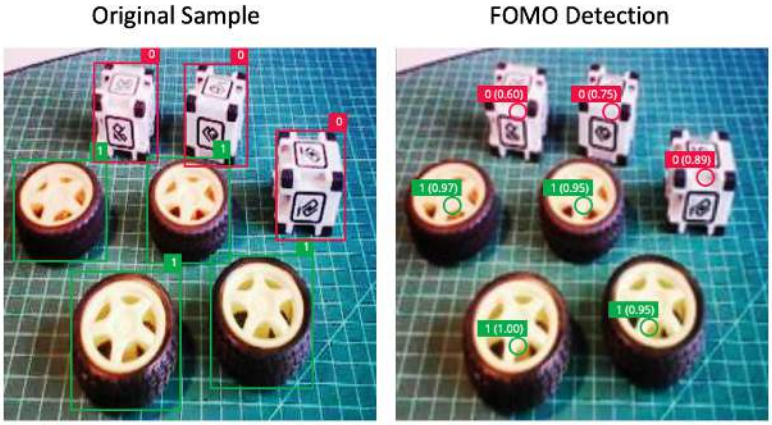
INFERENCING TIME
8 ms.

PEAK RAM USAGE
626.5K

FLASH USAGE
79.0K

Note that FOMO automatically added a third label background to the two previously defined *boxes* (0) and *wheels* (1).

On the `Model testing` tab, we can see that the accuracy was 94%. Here is one of the test sample results:



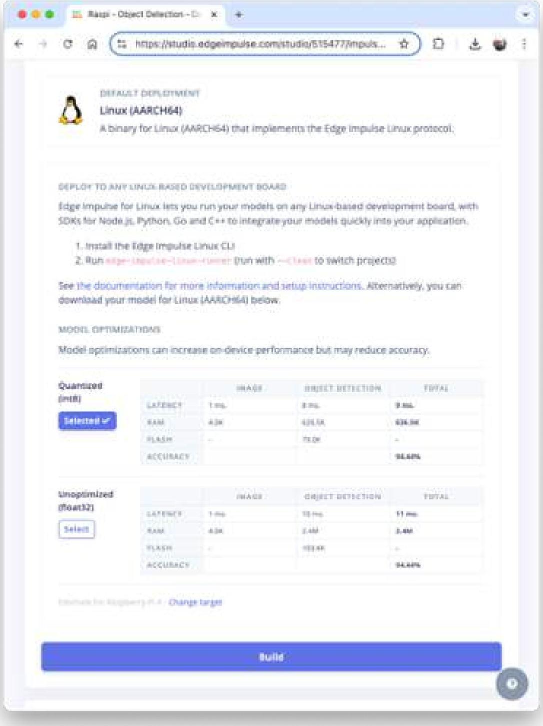
In object detection tasks, accuracy is generally not the primary evaluation metric. Object detection involves classifying objects and providing bounding boxes around them, making it a more complex problem than simple classification. The issue is that we do not have the bounding box, only the centroids. In short, using accuracy as a metric could be misleading and may not provide a complete understanding of how well the model is performing.

Deploying the model

As we did in the previous section, we can deploy the trained model as TFLite or Linux (AARCH64). Let's do it now as **Linux (AARCH64)**, a binary that implements the Edge Impulse Linux protocol.

Edge Impulse for Linux models is delivered in `.eim` format. This executable contains our "full impulse" created in Edge Impulse Studio. The impulse consists of the signal processing block(s) and any learning and anomaly block(s) we added and trained. It is compiled with optimizations for our processor or GPU (e.g., NEON instructions on ARM cores), plus a straightforward IPC layer (over a Unix socket).

At the `Deploy` tab, select the option `Linux (AARCH64)`, the `int8`model and press `Build`.



Raspi - Object Detection - E

https://studio.edgeimpulse.com/studio/515477/impuls...

DEFAULT DEPLOYMENT

Linux (AARCH64)

A binary for Linux (AARCH64) that implements the Edge Impulse Linux protocol.

DEPLOY TO ANY LINUX-BASED DEVELOPMENT BOARD

Edge Impulse for Linux lets you run your models on any Linux-based development board, with SDKs for Node.js, Python, Go and C++ to integrate your models quickly into your application.

1. Install the Edge Impulse Linux CLI
2. Run edge-impulse-linux-runner (run with --clean to switch projects)

See the documentation for more information and setup instructions. Alternatively, you can download your model for Linux (AARCH64) below.

MODEL OPTIMIZATIONS

Model optimizations can increase on-device performance but may reduce accuracy.

Quantized (int8)

Selected ✓

	IMAGE	OBJECT DETECTION	TOTAL
LATENCY	1 ms.	8 ms.	9 ms.
RAM	4.0K	625.5K	626.0K
FLASH	-	79.0K	-
ACCURACY			94.60%

Unoptimized (float32)

Select

	IMAGE	OBJECT DETECTION	TOTAL
LATENCY	1 ms.	10 ms.	11 ms.
RAM	4.0K	2.4M	2.4M
FLASH	-	153.4K	-
ACCURACY			94.40%

Estimate for Raspberry Pi 4 · Change target

The model will be automatically downloaded to your computer.

On our Raspi, let's create a new working area:

```
cd ~
cd Documents
mkdir EI_Linux
cd EI_Linux
mkdir models
mkdir images
```

Rename the model for easy identification:

For example, `raspi-object-detection-linux-aarch64-FOMO-int8.eim` and transfer it to the new Raspi folder `./models` and capture or get some images for inference and save them in the folder `./images`.

Inference and Post-Processing

The inference will be made using the Linux Python SDK. This library lets us run machine learning models and collect sensor data on Linux machines using Python. The SDK is open source and hosted on GitHub: edgeimpulse/linux-sdk-python.

Let's set up a Virtual Environment for working with the Linux Python SDK

```
python3 -m venv ~/eilinux
source ~/eilinux/bin/activate
```

And Install the all the libraries needed:

```
sudo apt-get update
sudo apt-get install libatlas-base-dev\
                     libportaudio0 libportaudio2
sudo apt-get installlibportaudiocpp0 portaudio19-dev

pip3 install edge_impulse_linux -i https://pypi.python.org/simple
pip3 install Pillow matplotlib pyaudio opencv-contrib-python

sudo apt-get install portaudio19-dev
pip3 install pyaudio
pip3 install opencv-contrib-python
```

Permit our model to be executable.

```
chmod +x raspi-object-detection-linux-aarch64-FOMO-int8.eim
```

Install the Jupiter Notebook on the new environment

```
pip3 install jupyter
```

Run a notebook locally (on the Raspi-4 or 5 with desktop)

```
jupyter notebook
```

or on the browser on your computer:

```
jupyter notebook --ip=192.168.4.210 --no-browser
```

Let's start a new notebook by following all the steps to detect cubes and wheels on an image using the FOMO model and the Edge Impulse Linux Python SDK.

Import the needed libraries:

```
import sys, time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image
import cv2
from edge_impulse_linux.image import ImageImpulseRunner
```

Define the model path and labels:

```
model_file = "raspi-object-detection-linux-aarch64-int8.eim"
model_path = "models/"+ model_file # Trained ML model from
                                   # Edge Impulse
labels = ['box', 'wheel']
```

> Remember that the model will output the class ID as values (0 and 1), following an alphabetic order regarding the class names.

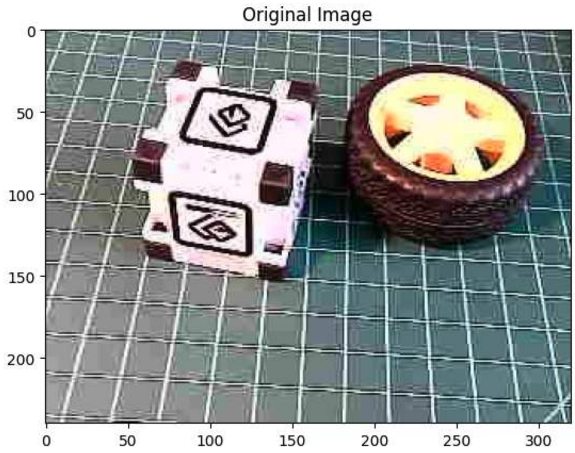
Load and initialize the model:

```
# Load the model file
runner = ImageImpulseRunner(model_path)

# Initialize model
model_info = runner.init()
```

The `model_info` will contain critical information about our model. However, unlike the TFLite interpreter, the EI Linux Python SDK library will now prepare the model for inference.

So, let's open the image and show it (Now, for compatibility, we will use OpenCV, the CV Library used internally by EI. OpenCV reads the image as BGR, so we will need to convert it to RGB :

```
# Load the image
img_path = "./images/1_box_1_wheel.jpg"
orig_img = cv2.imread(img_path)
img_rgb = cv2.cvtColor(orig_img, cv2.COLOR_BGR2RGB)

# Display the image
plt.imshow(img_rgb)
plt.title("Original Image")
plt.show()
```



Now we will get the features and the preprocessed image (`cropped`) using the `runner`:

```
features, cropped = runner.\
    get_features_from_image_auto_studio_settings(img_rgb)
```

And perform the inference. Let's also calculate the latency of the model:

```
res = runner.classify(features)
```

Let's get the output classes of objects detected, their bounding boxes centroids, and probabilities.

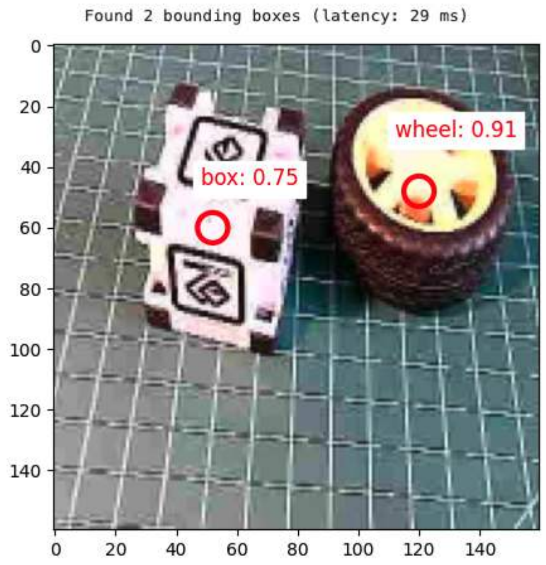
```
print('Found %d bounding boxes (%d ms.)' % (
  len(res["result"]["bounding_boxes"]),
  res['timing']['dsp'] + res['timing']['classification']))
for bb in res["result"]["bounding_boxes"]:
    print('\t%s (%.2f): x=%d y=%d w=%d h=%d' % (
      bb['label'], bb['value'], bb['x'],
      bb['y'], bb['width'], bb['height']))
```

```
Found 2 bounding boxes (29 ms.)
    1 (0.91): x=112 y=40 w=16 h=16
    0 (0.75): x=48 y=56 w=8 h=8
```

The results show that two objects were detected: one with class ID 0 (`box`) and one with class ID 1 (`wheel`), which is correct!

Let's visualize the result (The `threshold` is 0.5, the default value set during the model testing on the Edge Impulse Studio).

```
print('\tFound %d bounding boxes (latency: %d ms)' % (
  len(res["result"]["bounding_boxes"]),
  res['timing']['dsp'] + res['timing']['classification']))
plt.figure(figsize=(5,5))
plt.imshow(cropped)

# Go through each of the returned bounding boxes
bboxes = res['result']['bounding_boxes']
for bbox in bboxes:

    # Get the corners of the bounding box
    left = bbox['x']
    top = bbox['y']
    width = bbox['width']
    height = bbox['height']

    # Draw a circle centered on the detection
    circ = plt.Circle((left+width//2, top+height//2), 5,
                     fill=False, color='red', linewidth=3)
    plt.gca().add_patch(circ)
    class_id = int(bbox['label'])
    class_name = labels[class_id]
    plt.text(left, top-10, f'{class_name}: {bbox["value"]:.2f}',
              color='red', fontsize=12, backgroundcolor='white')
plt.show()
```



Exploring a YOLO Model using Ultralitics

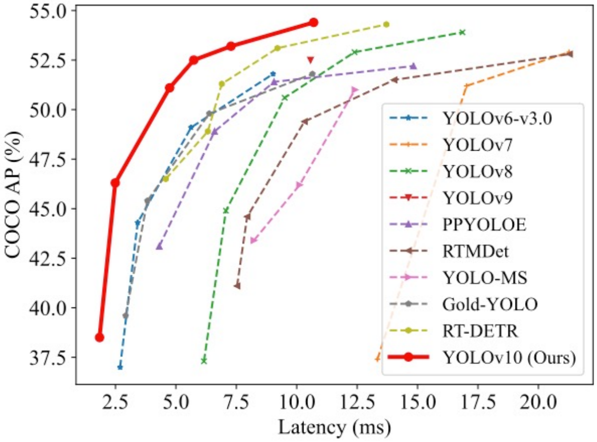
For this lab, we will explore YOLOv8. Ultralytics YOLOv8 is a version of the acclaimed real-time object detection and image segmentation model, YOLO. YOLOv8 is built on cutting-edge advancements in deep learning and computer vision, offering unparalleled performance in terms of speed and accuracy. Its streamlined design makes it suitable for various applications and easily adaptable to different hardware platforms, from edge devices to cloud APIs.

Talking about the YOLO Model

The YOLO (You Only Look Once) model is a highly efficient and widely used object detection algorithm known for its real-time processing capabilities. Unlike traditional object detection systems that repurpose classifiers or localizers to perform detection, YOLO frames the detection problem as a single regression task. This innovative approach enables YOLO to simultaneously predict multiple bounding boxes and their class probabilities from full images in one evaluation, significantly boosting its speed.

Key Features:

1. **Single Network Architecture**:
   - YOLO employs a single neural network to process the entire image. This network divides the image into a grid and, for each grid cell, directly predicts bounding boxes and associated class probabilities. This end-to-end training improves speed and simplifies the model architecture.
2. **Real-Time Processing**:
   - One of YOLO's standout features is its ability to perform object detection in real-time. Depending on the version and hardware, YOLO can process images at high frames per second (FPS). This makes it ideal for applications requiring quick and accurate object detection, such as video surveillance, autonomous driving, and live sports analysis.
3. **Evolution of Versions**:
   - Over the years, YOLO has undergone significant improvements, from YOLOv1 to the latest YOLOv10. Each iteration has introduced enhancements in accuracy, speed, and efficiency. YOLOv8, for instance, incorporates advancements in network architecture, improved training methodologies, and better support for various hardware, ensuring a more robust performance.
   - Although YOLOv10 is the family's newest member with an encouraging performance based on its paper, it was just released (May 2024) and is not fully integrated with the Ultralitycs library. Conversely, the precision-recall curve analysis suggests that YOLOv8 generally outperforms YOLOv9, capturing a higher proportion of true positives while minimizing false positives more effectively (for more details, see this article). So, this lab is based on the YOLOv8n.



4. **Accuracy and Efficiency**:

   - While early versions of YOLO traded off some accuracy for speed, recent versions have made substantial strides in balancing both. The newer models are faster and more accurate, detecting small objects (such as bees) and performing well on complex datasets.

5. **Wide Range of Applications**:

   - YOLO's versatility has led to its adoption in numerous fields. It is used in traffic monitoring systems to detect and count vehicles, security applications to identify potential threats and agricultural technology to monitor crops and livestock. Its application extends to any domain requiring efficient and accurate object detection.

6. **Community and Development**:

   - YOLO continues to evolve and is supported by a strong community of developers and researchers (being the YOLOv8 very strong). Open-source implementations and extensive documentation have made it accessible for customization and integration into various projects. Popular deep learning frameworks like Darknet, TensorFlow, and PyTorch support YOLO, further broadening its applicability.
   - Ultralitics YOLOv8 can not only Detect (our case here) but also Segment and Pose models pre-trained on the COCO dataset and YOLOv8 Classify models pre-trained on the ImageNet dataset. Track mode is available for all Detect, Segment, and Pose models.

Figure 20.20: Ultralytics YOLO supported tasks

Installation

On our Raspi, let's deactivate the current environment to create a new working area:

```
deactivate
cd ~
cd Documents/
mkdir YOLO
cd YOLO
mkdir models
mkdir images
```

Let's set up a Virtual Environment for working with the Ultralytics YOLOv8

```
python3 -m venv ~/yolo
source ~/yolo/bin/activate
```

And install the Ultralytics packages for local inference on the Raspi

1. Update the packages list, install pip, and upgrade to the latest:

```
sudo apt update
sudo apt install python3-pip -y
pip install -U pip
```

2. Install the `ultralytics` pip package with optional dependencies:

```
pip install ultralytics[export]
```

3. Reboot the device:

```
sudo reboot
```

Testing the YOLO

After the Raspi-Zero booting, let's activate the `yolo` env, go to the working directory,

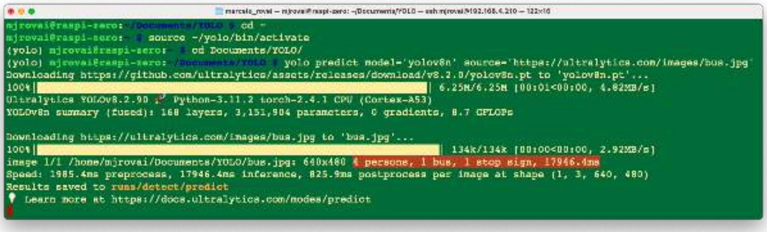
```
source ~/yolo/bin/activate
cd /Documents/YOLO
```

and run inference on an image that will be downloaded from the Ultralytics website, using the YOLOV8n model (the smallest in the family) at the Terminal (CLI):

```
yolo predict model='yolov8n' \
     source='https://ultralytics.com/images/bus.jpg'
```

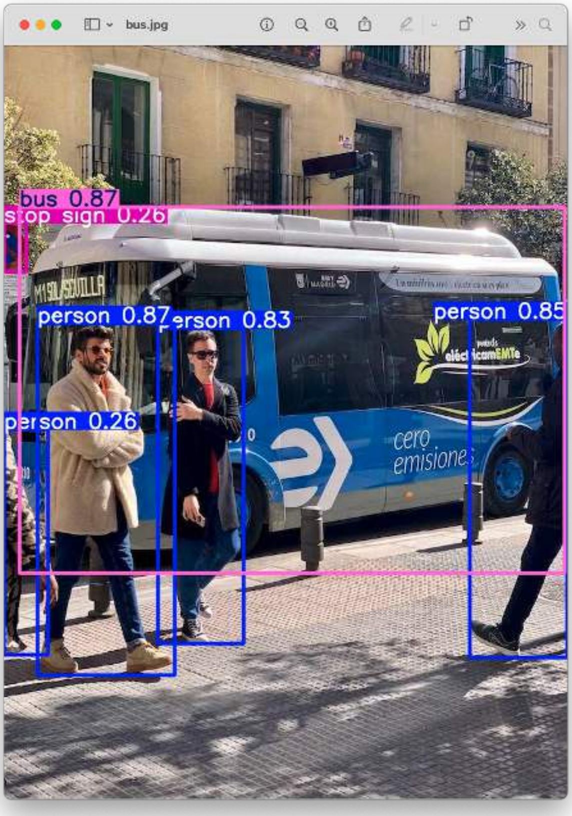
The YOLO model family is pre-trained with the COCO dataset.

The inference result will appear in the terminal. In the image (bus.jpg), 4 `persons`, 1 `bus`, and 1 `stop signal` were detected:


marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 122×16

```
mjrovai@raspi-zero:~/Documents/YOLO $ cd ~
mjrovai@raspi-zero:~ $ source ~/yolo/bin/activate
(yolo) mjrovai@raspi-zero:~ $ cd Documents/YOLO/
(yolo) mjrovai@raspi-zero:~/Documents/YOLO $ yolo predict model='yolov8n' source='https://ultralytics.com/images/bus.jpg'
Downloading https://github.com/ultralytics/assets/releases/download/v8.2.0/yolov8n.pt to 'yolov8n.pt'...
100%|████████████████████████████████████████| 6.25M/6.25M [00:01<00:00, 4.82MB/s]
Ultralytics YOLOv8.2.90 🚀 Python-3.11.2 torch-2.4.1 CPU (Cortex-A53)
YOLOv8n summary (fused): 168 layers, 3,151,904 parameters, 0 gradients, 8.7 GFLOPs

Downloading https://ultralytics.com/images/bus.jpg to 'bus.jpg'...
100%|████████████████████████████████████████| 134k/134k [00:00<00:00, 2.92MB/s]
image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x480 4 persons, 1 bus, 1 stop sign, 17946.4ms
Speed: 1985.4ms preprocess, 17946.4ms inference, 825.9ms postprocess per image at shape (1, 3, 640, 480)
Results saved to runs/detect/predict
💡 Learn more at https://docs.ultralytics.com/modes/predict
```

Also, we got a message that `Results saved to runs/detect/predict`. Inspecting that directory, we can see a new image saved (bus.jpg). Let's download it from the Raspi-Zero to our desktop for inspection:



So, the Ultrayitics YOLO is correctly installed on our Raspi. But, on the Raspi-Zero, an issue is the high latency for this inference, around 18 seconds, even with the most miniature model of the family (YOLOv8n).

Export Model to NCNN format

Deploying computer vision models on edge devices with limited computational power, such as the Raspi-Zero, can cause latency issues. One alternative is to use a format optimized for optimal performance. This ensures that even devices with limited processing power can handle advanced computer vision tasks well.

Of all the model export formats supported by Ultralytics, the NCNN is a high-performance neural network inference computing framework optimized for mobile platforms. From the beginning of the design, NCNN was deeply considerate about deployment and use on mobile phones and did not have third-party dependencies. It is cross-platform and runs faster than all known open-source frameworks (such as TFLite).

NCNN delivers the best inference performance when working with Raspberry Pi devices. NCNN is highly optimized for mobile embedded platforms (such as ARM architecture).

So, let's convert our model and rerun the inference:

1. Export a YOLOv8n PyTorch model to NCNN format, creating: '/yolov8n_-ncnn_model'

```
yolo export model=yolov8n.pt format=ncnn
```

2. Run inference with the exported model (now the source could be the bus.jpg image that was downloaded from the website to the current directory on the last inference):

```
yolo predict model='./yolov8n_ncnn_model' source='bus.jpg'
```

The first inference, when the model is loaded, usually has a high latency (around 17s), but from the 2nd, it is possible to note that the inference goes down to around 2s.

Exploring YOLO with Python

To start, let's call the Python Interpreter so we can explore how the YOLO model works, line by line:

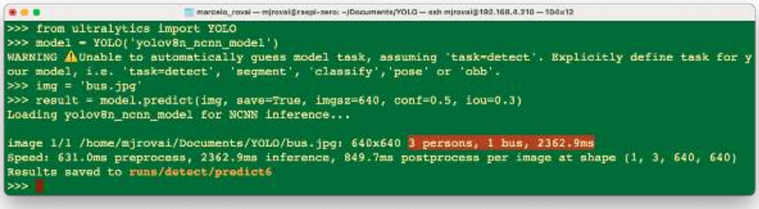
```
python3
```

Now, we should call the YOLO library from Ultralitics and load the model:

```
from ultralytics import YOLO
model = YOLO('yolov8n_ncnn_model')
```

Next, run inference over an image (let's use again `bus.jpg`):

```
img = 'bus.jpg'
result = model.predict(img, save=True, imgsz=640, conf=0.5,
                       iou=0.3)
```



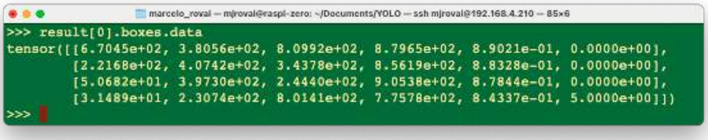
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 100x12

```
>>> from ultralytics import YOLO
>>> model = YOLO('yolov8n_ncnn_model')
WARNING ⚠️Unable to automatically guess model task, assuming 'task=detect'. Explicitly define task for y
our model, i.e. 'task=detect', 'segment', 'classify','pose' or 'obb'.
>>> img = 'bus.jpg'
>>> result = model.predict(img, save=True, imgsz=640, conf=0.5, iou=0.3)
Loading yolov8n_ncnn_model for NCNN inference...

image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 2362.9ms
Speed: 631.0ms preprocess, 2362.9ms inference, 849.7ms postprocess per image at shape (1, 3, 640, 640)
Results saved to runs/detect/predict6
>>>
```

We can verify that the result is almost identical to the one we get running the inference at the terminal level (CLI), except that the bus stop was not detected with the reduced NCNN model. Note that the latency was reduced.

Let's analyze the "result" content.

For example, we can see `result[0].boxes.data`, showing us the main inference result, which is a tensor shape (4, 6). Each line is one of the objects detected, being the 4 first columns, the bounding boxes coordinates, the 5th, the confidence, and the 6th, the class (in this case, `0: person` and `5: bus`):



marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 85×6

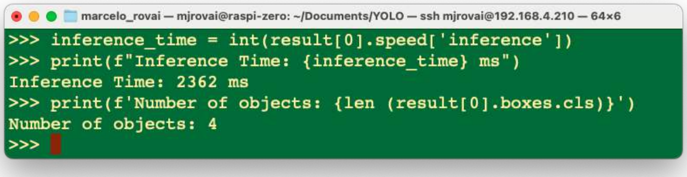
```
>>> result[0].boxes.data
tensor([[6.7045e+02, 3.8056e+02, 8.0992e+02, 8.7965e+02, 8.9021e-01, 0.0000e+00],
        [2.2168e+02, 4.0742e+02, 3.4378e+02, 8.5619e+02, 8.8328e-01, 0.0000e+00],
        [5.0682e+01, 3.9730e+02, 2.4440e+02, 9.0538e+02, 8.7844e-01, 0.0000e+00],
        [3.1489e+01, 2.3074e+02, 8.0141e+02, 7.7578e+02, 8.4337e-01, 5.0000e+00]])
>>>
```

We can access several inference results separately, as the inference time, and have it printed in a better format:

```
inference_time = int(result[0].speed['inference'])
print(f"Inference Time: {inference_time} ms")
```

Or we can have the total number of objects detected:

```
print(f'Number of objects: {len (result[0].boxes.cls)}')
```



marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 64×6

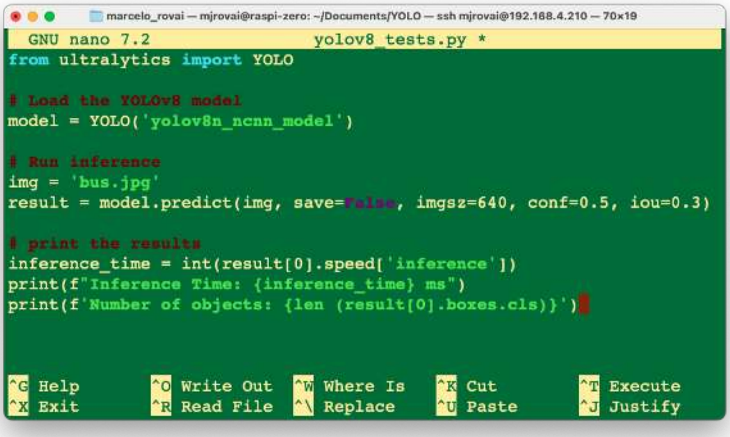
```
>>> inference_time = int(result[0].speed['inference'])
>>> print(f"Inference Time: {inference_time} ms")
Inference Time: 2362 ms
>>> print(f'Number of objects: {len (result[0].boxes.cls)}')
Number of objects: 4
>>>
```

With Python, we can create a detailed output that meets our needs (See Model Prediction with Ultralytics YOLO for more details). Let's run a Python script instead of manually entering it line by line in the interpreter, as shown below. Let's use `nano` as our text editor. First, we should create an empty Python script named, for example, `yolov8_tests.py`:

```
nano yolov8_tests.py
```

Enter with the code lines:

```python
from ultralytics import YOLO

# Load the YOLOv8 model
model = YOLO('yolov8n_ncnn_model')

# Run inference
img = 'bus.jpg'
result = model.predict(img, save=False, imgsz=640,
                       conf=0.5, iou=0.3)

# print the results
inference_time = int(result[0].speed['inference'])
print(f"Inference Time: {inference_time} ms")
print(f'Number of objects: {len (result[0].boxes.cls)}')
```



```python
from ultralytics import YOLO

# Load the YOLOv8 model
model = YOLO('yolov8n_ncnn_model')

# Run inference
img = 'bus.jpg'
result = model.predict(img, save=False, imgsz=640, conf=0.5, iou=0.3)

# print the results
inference_time = int(result[0].speed['inference'])
print(f"Inference Time: {inference_time} ms")
print(f'Number of objects: {len (result[0].boxes.cls)}')
```

And enter with the commands: [CTRL+O] + [ENTER] +[CTRL+X] to save the Python script.

Run the script:

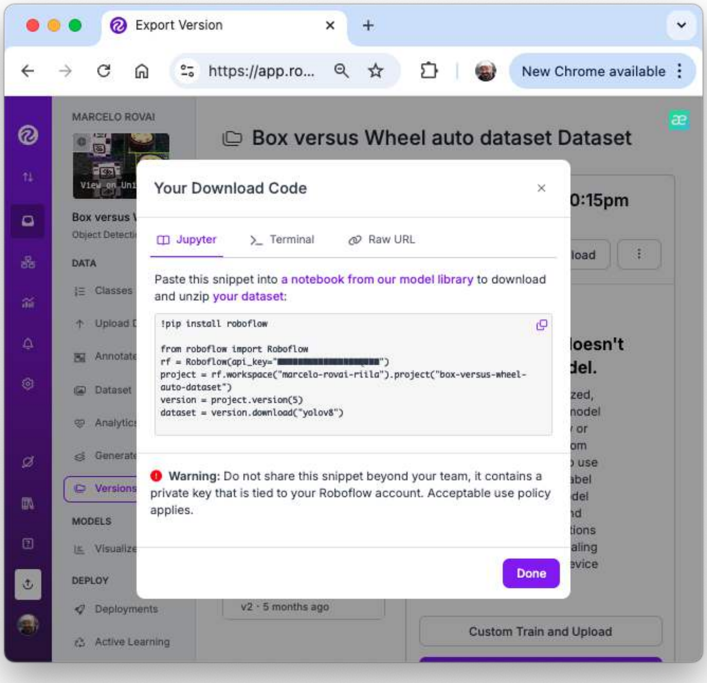
```
python yolov8_tests.py
```

The result is the same as running the inference at the terminal level (CLI) and with the built-in Python interpreter.

> Calling the YOLO library and loading the model for inference for the first time takes a long time, but the inferences after that will be much faster. For example, the first single inference can take several seconds, but after that, the inference time should be reduced to less than 1 second.

Training YOLOv8 on a Customized Dataset

Return to our "Box versus Wheel" dataset, labeled on Roboflow. On the `Download Dataset`, instead of `Download a zip to computer` option done for training on Edge Impulse Studio, we will opt for `Show download code`. This option will open a pop-up window with a code snippet that should be pasted into our training notebook.



Your Download Code

Paste this snippet into a notebook from our model library to download and unzip your dataset:

```
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="[redacted]")
project = rf.workspace("marcelo-rovai-riila").project("box-versus-wheel-
auto-dataset")
version = project.version(5)
dataset = version.download("yolov8")
```

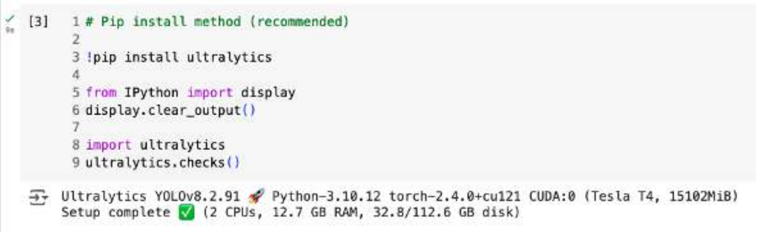
Warning: Do not share this snippet beyond your team, it contains a private key that is tied to your Roboflow account. Acceptable use policy applies.

For training, let's adapt one of the public examples available from Ultralitytics and run it on Google Colab. Below, you can find mine to be adapted in your project:

- YOLOv8 Box versus Wheel Dataset Training [Open In Colab]

Critical points on the Notebook:

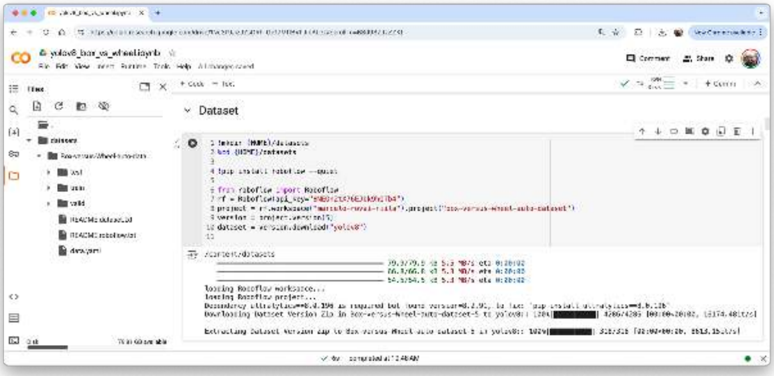
1. Run it with GPU (the NVidia T4 is free)
2. Install Ultralytics using PIP.



```
[3]
# Pip install method (recommended)

!pip install ultralytics

from IPython import display
display.clear_output()

import ultralytics
ultralytics.checks()
```

```
Ultralytics YOLOv8.2.91 🚀 Python-3.10.12 torch-2.4.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
Setup complete ✅ (2 CPUs, 12.7 GB RAM, 32.8/112.6 GB disk)
```

3. Now, you can import the YOLO and upload your dataset to the CoLab, pasting the Download code that we get from Roboflow. Note that our dataset will be mounted under `/content/datasets/`:



yolov8_box_vs_wheel.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
Files
+ Code + Text
Dataset
datasets
Box-versus-Wheel-auto-dataset-5
test
train
valid
README.dataset.txt
README.roboflow.txt
data.yaml

```
!mkdir {HOME}/datasets
%cd {HOME}/datasets

!pip install roboflow --quiet

from roboflow import Roboflow
rf = Roboflow(api_key="[illegible]")
project = rf.workspace("marcelo-rovai-riila").project("box-versus-wheel-auto-dataset")
version = project.version(5)
dataset = version.download("yolov8")
```

```
/content/datasets
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 79.3/79.5 kB 5.3 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 66.8/66.8 kB 5.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 54.5/54.5 kB 5.3 MB/s eta 0:00:00
loading Roboflow workspace...
loading Roboflow project...
Dependency ultralytics==8.0.196 is required but found version=8.2.91, to fix: `pip install ultralytics==8.0.196`
Downloading Dataset Version Zip in Box-versus-Wheel-auto-dataset-5 to yolov8:: 100%|██████████| 4286/4285 [00:00<00:00, 15174.48it/s]

Extracting Dataset Version Zip to Box-versus-Wheel-auto-dataset-5 in yolov8:: 100%|██████████| 318/318 [00:00<00:00, 8513.15it/s]
```

4. It is essential to verify and change the file `data.yaml` with the correct path for the images (copy the path on each `images` folder).

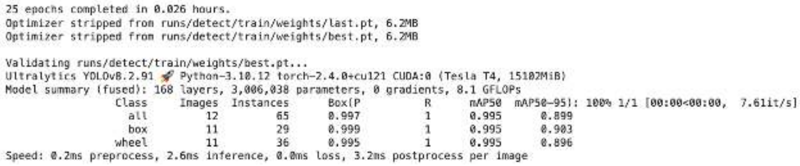
```
names:
- box
- wheel
nc: 2
roboflow:
  license: CC BY 4.0
  project: box-versus-wheel-auto-dataset
  url: https://universe.roboflow.com/marcelo-rovai-riila/ \
     box-versus-wheel-auto-dataset/dataset/5
  version: 5
  workspace: marcelo-rovai-riila
test: /content/datasets/Box-versus-Wheel-auto-dataset-5/ \
      test/images
train: /content/datasets/Box-versus-Wheel-auto-dataset-5/ \
       train/images
val: /content/datasets/Box-versus-Wheel-auto-dataset-5/ \
     valid/images
```

5. Define the main hyperparameters that you want to change from default, for example:

```
MODEL = 'yolov8n.pt'
IMG_SIZE = 640
EPOCHS = 25 # For a final project, you should consider
            # at least 100 epochs
```

6. Run the training (using CLI):

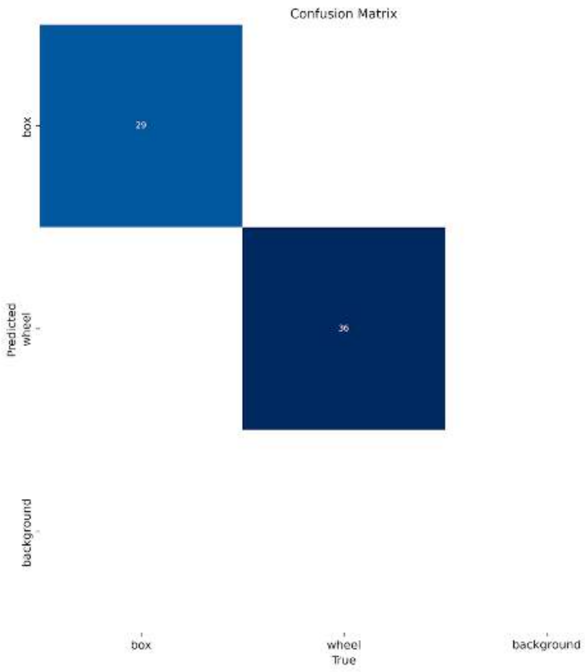
```
!yolo task=detect mode=train model={MODEL} \
  data={dataset.location}/data.yaml \
  epochs={EPOCHS}
  imgsz={IMG_SIZE} plots=True
```



```
25 epochs completed in 0.026 hours.
Optimizer stripped from runs/detect/train/weights/last.pt, 6.2MB
Optimizer stripped from runs/detect/train/weights/best.pt, 6.2MB

Validating runs/detect/train/weights/best.pt...
Ultralytics YOLOv8.2.91 🚀 Python-3.10.12 torch-2.4.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 3,006,038 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 1/1 [00:00<00:00,  7.61it/s]
                   all         12         65      0.997          1      0.995      0.899
                   box         11         29      0.999          1      0.995      0.903
                 wheel         11         36      0.995          1      0.995      0.896
Speed: 0.2ms preprocess, 2.6ms inference, 0.0ms loss, 3.2ms postprocess per image
```

Figure 20.21: image-20240910111319804

The model took a few minutes to be trained and has an excellent result (mAP50 of 0.995). At the end of the training, all results are saved in the folder listed, for example: `/runs/detect/train/`. There, you can find, for example, the confusion matrix.



7. Note that the trained model (`best.pt`) is saved in the folder `/runs/detect/train/weights/`. Now, you should validate the trained model with the `valid/images`.

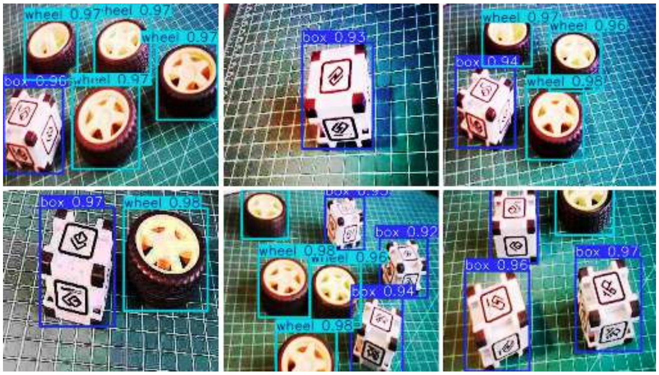
```
!yolo task=detect mode=val model={HOME}/runs/detect/train/\
       weights/best.pt data={dataset.location}/data.yaml
```

The results were similar to training.

8. Now, we should perform inference on the images left aside for testing

```
!yolo task=detect mode=predict model={HOME}/runs/detect/train/\
     weights/best.pt conf=0.25 source={dataset.location}/test/\
     images save=True
```

The inference results are saved in the folder `runs/detect/predict`. Let's see some of them:



9. It is advised to export the train, validation, and test results for a Drive at Google. To do so, we should mount the drive.

```
    from google.colab import drive
    drive.mount('/content/gdrive')
```

   and copy the content of `/runs` folder to a folder that you should create in your Drive, for example:

```
    !scp -r /content/runs '/content/gdrive/MyDrive/\
         10_UNIFEI/Box_vs_Wheel_Project'
```

Inference with the trained model, using the Raspi

Download the trained model `/runs/detect/train/weights/best.pt` to your computer. Using the FileZilla FTP, let's transfer the `best.pt` to the Raspi models folder (before the transfer, you may change the model name, for example, `box_wheel_320_yolo.pt`).

Using the FileZilla FTP, let's transfer a few images from the test dataset to `.\YOLO\images`:

Let's return to the YOLO folder and use the Python Interpreter:

```
cd ..
python
```

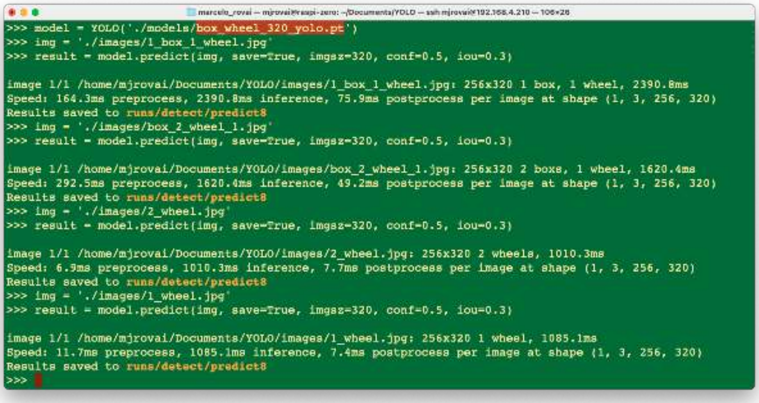
As before, we will import the YOLO library and define our converted model to detect bees:

```
from ultralytics import YOLO
model = YOLO('./models/box_wheel_320_yolo.pt')
```

Now, let's define an image and call the inference (we will save the image result this time to external verification):

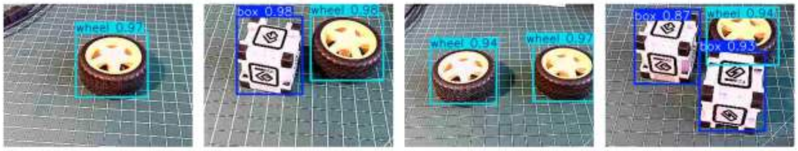
```python
img = './images/1_box_1_wheel.jpg'
result = model.predict(img, save=True, imgsz=320,
                       conf=0.5, iou=0.3)
```

Let's repeat for several images. The inference result is saved on the variable `result`, and the processed image on `runs/detect/predict8`


marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 106×28

```
>>> model = YOLO('./models/box_wheel_320_yolo.pt')
>>> img = './images/1_box_1_wheel.jpg'
>>> result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/images/1_box_1_wheel.jpg: 256x320 1 box, 1 wheel, 2390.8ms
Speed: 164.3ms preprocess, 2390.8ms inference, 75.9ms postprocess per image at shape (1, 3, 256, 320)
Results saved to runs/detect/predict8
>>> img = './images/box_2_wheel_1.jpg'
>>> result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/images/box_2_wheel_1.jpg: 256x320 2 boxs, 1 wheel, 1620.4ms
Speed: 292.5ms preprocess, 1620.4ms inference, 49.2ms postprocess per image at shape (1, 3, 256, 320)
Results saved to runs/detect/predict8
>>> img = './images/2_wheel.jpg'
>>> result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/images/2_wheel.jpg: 256x320 2 wheels, 1010.3ms
Speed: 6.9ms preprocess, 1010.3ms inference, 7.7ms postprocess per image at shape (1, 3, 256, 320)
Results saved to runs/detect/predict8
>>> img = './images/1_wheel.jpg'
>>> result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/images/1_wheel.jpg: 256x320 1 wheel, 1085.1ms
Speed: 11.7ms preprocess, 1085.1ms inference, 7.4ms postprocess per image at shape (1, 3, 256, 320)
Results saved to runs/detect/predict8
>>>
```

Using FileZilla FTP, we can send the inference result to our Desktop for verification:



We can see that the inference result is excellent! The model was trained based on the smaller base model of the YOLOv8 family (YOLOv8n). The issue is the latency, around 1 second (or 1 FPS on the Raspi-Zero). Of course, we can reduce this latency and convert the model to TFLite or NCNN.

Object Detection on a live stream

All the models explored in this lab can detect objects in real-time using a camera. The captured image should be the input for the trained and converted model. For the Raspi-4 or 5 with a desktop, OpenCV can capture the frames and display the inference result.

However, creating a live stream with a webcam to detect objects in real-time is also possible. For example, let's start with the script developed for the Image Classification app and adapt it for a *Real-Time Object Detection Web Application Using TensorFlow Lite and Flask*.

This app version will work for all TFLite models. Verify if the model is in its correct folder, for example:

```
model_path = "./models/ssd-mobilenet-v1-tflite-default-v1.tflite"
```

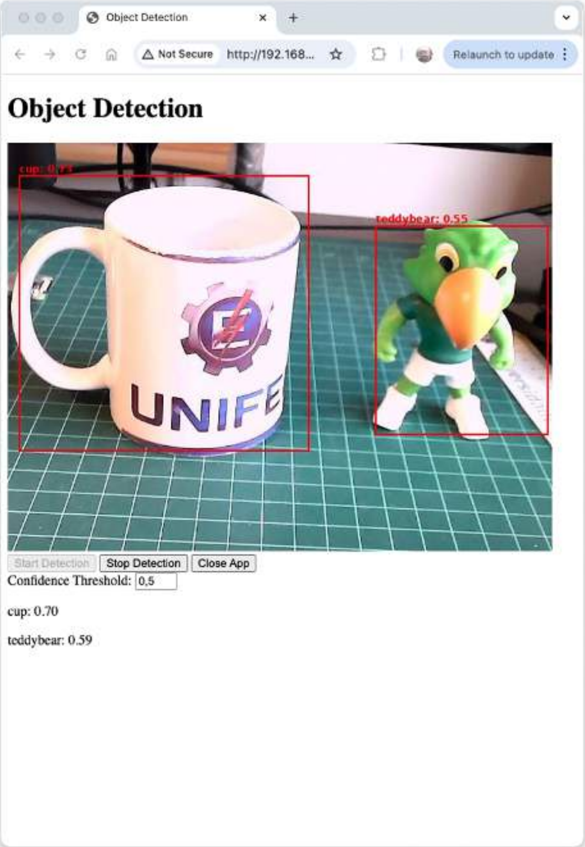
Download the Python script `object_detection_app.py` from GitHub.
And on the terminal, run:

```
python3 object_detection_app.py
```

And access the web interface:

- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`
- From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace `<raspberry_pi_ip>` with your Raspberry Pi's IP address). For example: `http://192.168.4.210:5000/`

Here are some screenshots of the app running on an external desktop



Object Detection

Confidence Threshold: 0,5

cup: 0.70

teddybear: 0.59

Let's see a technical description of the key modules used in the object detection application:

1. **TensorFlow Lite (tflite_runtime)**:
    - Purpose: Efficient inference of machine learning models on edge devices.
    - Why: TFLite offers reduced model size and optimized performance compared to full TensorFlow, which is crucial for resource-constrained devices like Raspberry Pi. It supports hardware acceleration and quantization, further improving efficiency.
    - Key functions: `Interpreter` for loading and running the model, `get_input_details()`, and `get_output_details()` for interfacing with the model.

2. **Flask**:
    - Purpose: Lightweight web framework for creating the backend server.
    - Why: Flask's simplicity and flexibility make it ideal for rapidly developing and deploying web applications. It's less resource-intensive than larger frameworks suitable for edge devices.
    - Key components: route decorators for defining API endpoints, `Response` objects for streaming video, `render_template_string` for serving dynamic HTML.

3. **Picamera2**:
    - Purpose: Interface with the Raspberry Pi camera module.
    - Why: Picamera2 is the latest library for controlling Raspberry Pi cameras, offering improved performance and features over the original Picamera library.
    - Key functions: `create_preview_configuration()` for setting up the camera, `capture_file()` for capturing frames.

4. **PIL (Python Imaging Library)**:
    - Purpose: Image processing and manipulation.
    - Why: PIL provides a wide range of image processing capabilities. It's used here to resize images, draw bounding boxes, and convert between image formats.
    - Key classes: `Image` for loading and manipulating images, `ImageDraw` for drawing shapes and text on images.

5. **NumPy**:
    - Purpose: Efficient array operations and numerical computing.
    - Why: NumPy's array operations are much faster than pure Python lists, which is crucial for efficiently processing image data and model inputs/outputs.
    - Key functions: `array()` for creating arrays, `expand_dims()` for adding dimensions to arrays.

6. **Threading**:
   - Purpose: Concurrent execution of tasks.
   - Why: Threading allows simultaneous frame capture, object detection, and web server operation, crucial for maintaining real-time performance.
   - Key components: `Thread` class creates separate execution threads, and Lock is used for thread synchronization.
7. **io.BytesIO**:
   - Purpose: In-memory binary streams.
   - Why: Allows efficient handling of image data in memory without needing temporary files, improving speed and reducing I/O operations.
8. **time**:
   - Purpose: Time-related functions.
   - Why: Used for adding delays (`time.sleep()`) to control frame rate and for performance measurements.
9. **jQuery (client-side)**:
   - Purpose: Simplified DOM manipulation and AJAX requests.
   - Why: It makes it easy to update the web interface dynamically and communicate with the server without page reloads.
   - Key functions: `.get()` and `.post()` for AJAX requests, DOM manipulation methods for updating the UI.

Regarding the main app system architecture:

1. **Main Thread**: Runs the Flask server, handling HTTP requests and serving the web interface.
2. **Camera Thread**: Continuously captures frames from the camera.
3. **Detection Thread**: Processes frames through the TFLite model for object detection.
4. **Frame Buffer**: Shared memory space (protected by locks) storing the latest frame and detection results.

And the app data flow, we can describe in short:

1. Camera captures frame → Frame Buffer
2. Detection thread reads from Frame Buffer → Processes through TFLite model → Updates detection results in Frame Buffer
3. Flask routes access Frame Buffer to serve the latest frame and detection results
4. Web client receives updates via AJAX and updates UI

This architecture allows for efficient, real-time object detection while maintaining a responsive web interface running on a resource-constrained edge device like a Raspberry Pi. Threading and efficient libraries like TFLite and PIL enable the system to process video frames in real-time, while Flask and jQuery provide a user-friendly way to interact with them.

You can test the app with another pre-processed model, such as the EfficientDet, changing the app line:

```
model_path = "./models/lite-model_efficientdet_lite0_\
   detection_metadata_1.tflite"
```

> If we want to use the app for the SSD-MobileNetV2 model, trained on Edge Impulse Studio with the "Box versus Wheel" dataset, the code should also be adapted depending on the input details, as we have explored on its notebook.

Conclusion

This lab has explored the implementation of object detection on edge devices like the Raspberry Pi, demonstrating the power and potential of running advanced computer vision tasks on resource-constrained hardware. We've covered several vital aspects:

1. **Model Comparison**: We examined different object detection models, including SSD-MobileNet, EfficientDet, FOMO, and YOLO, comparing their performance and trade-offs on edge devices.
2. **Training and Deployment**: Using a custom dataset of boxes and wheels (labeled on Roboflow), we walked through the process of training models using Edge Impulse Studio and Ultralytics and deploying them on Raspberry Pi.
3. **Optimization Techniques**: To improve inference speed on edge devices, we explored various optimization methods, such as model quantization (TFLite int8) and format conversion (e.g., to NCNN).
4. **Real-time Applications**: The lab exemplified a real-time object detection web application, demonstrating how these models can be integrated into practical, interactive systems.
5. **Performance Considerations**: Throughout the lab, we discussed the balance between model accuracy and inference speed, a critical consideration for edge AI applications.

The ability to perform object detection on edge devices opens up numerous possibilities across various domains, from precision agriculture, industrial automation, and quality control to smart home applications and environmental monitoring. By processing data locally, these systems can offer reduced latency, improved privacy, and operation in environments with limited connectivity.

Looking ahead, potential areas for further exploration include:

- Implementing multi-model pipelines for more complex tasks
- Exploring hardware acceleration options for Raspberry Pi
- Integrating object detection with other sensors for more comprehensive edge AI systems

- Developing edge-to-cloud solutions that leverage both local processing and cloud resources

Object detection on edge devices can create intelligent, responsive systems that bring the power of AI directly into the physical world, opening up new frontiers in how we interact with and understand our environment.

Resources

- Dataset ("Box versus Wheel")
- SSD-MobileNet Notebook on a Raspi
- EfficientDet Notebook on a Raspi
- FOMO - EI Linux Notebook on a Raspi
- YOLOv8 Box versus Wheel Dataset Training on CoLab
- Edge Impulse Project - SSD MobileNet and FOMO
- Python Scripts
- Models

Small Language Models (SLM)

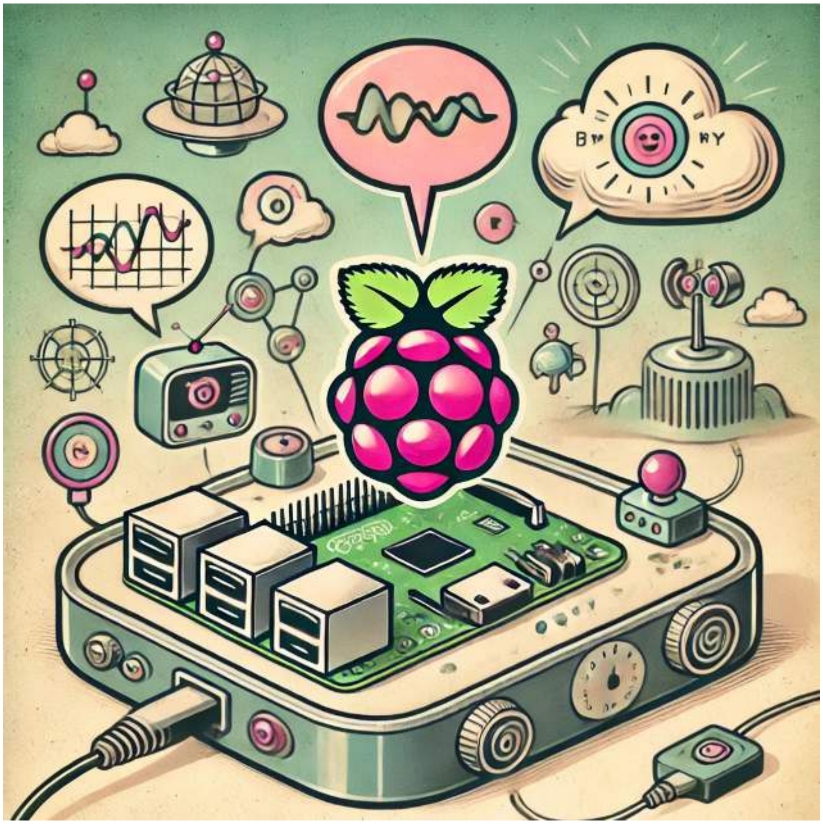


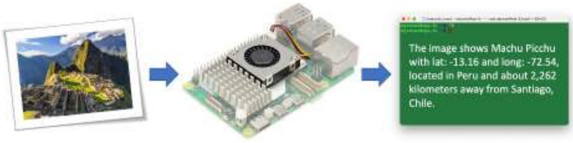
Figure 20.22: *DALL·E prompt - A 1950s-style cartoon illustration showing a Raspberry Pi running a small language model at the edge. The Raspberry Pi is stylized in a retro-futuristic way with rounded edges and chrome accents, connected to playful cartoonish sensors and devices. Speech bubbles are floating around, representing language processing, and the background has a whimsical landscape of interconnected devices with wires and small gadgets, all drawn in a vintage cartoon style. The color palette uses soft pastel colors and bold outlines typical of 1950s cartoons, giving a fun and nostalgic vibe to the scene.*

Overview

In the fast-growing area of artificial intelligence, edge computing presents an opportunity to decentralize capabilities traditionally reserved for powerful, centralized servers. This lab explores the practical integration of small versions of traditional large language models (LLMs) into a Raspberry Pi 5, transforming this edge device into an AI hub capable of real-time, on-site data processing.

As large language models grow in size and complexity, Small Language Models (SLMs) offer a compelling alternative for edge devices, striking a balance between performance and resource efficiency. By running these models directly on Raspberry Pi, we can create responsive, privacy-preserving applications that operate even in environments with limited or no internet connectivity.

This lab will guide you through setting up, optimizing, and leveraging SLMs on Raspberry Pi. We will explore the installation and utilization of Ollama. This open-source framework allows us to run LLMs locally on our machines (our desktops or edge devices such as the Raspberry Pis or NVidia Jetsons). Ollama is designed to be efficient, scalable, and easy to use, making it a good option for deploying AI models such as Microsoft Phi, Google Gemma, Meta Llama, and LLaVa (Multimodal). We will integrate some of those models into projects using Python's ecosystem, exploring their potential in real-world scenarios (or at least point in this direction).



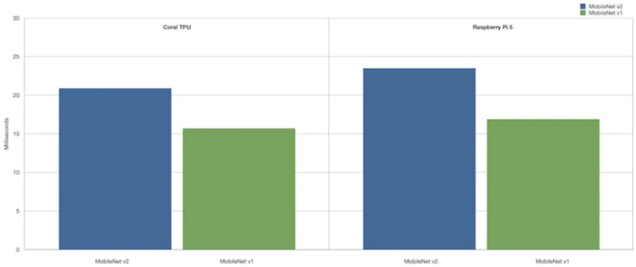
```
The image shows Machu Picchu
with lat: -13.16 and long: -72.54,
located in Peru and about 2,262
kilometers away from Santiago,
Chile.
```

Setup

We could use any Raspi model in the previous labs, but here, the choice must be the Raspberry Pi 5 (Raspi-5). It is a robust platform that substantially upgrades the last version 4, equipped with the Broadcom BCM2712, a 2.4 GHz quad-core 64-bit Arm Cortex-A76 CPU featuring Cryptographic Extension and enhanced caching capabilities. It boasts a VideoCore VII GPU, dual 4Kp60 HDMI® outputs with HDR, and a 4Kp60 HEVC decoder. Memory options include 4 GB and 8 GB of high-speed LPDDR4X SDRAM, with 8GB being our choice to run SLMs. It also features expandable storage via a microSD card slot and a PCIe 2.0 interface for fast peripherals such as M.2 SSDs (Solid State Drives).

> For real SSL applications, SSDs are a better option than SD cards.

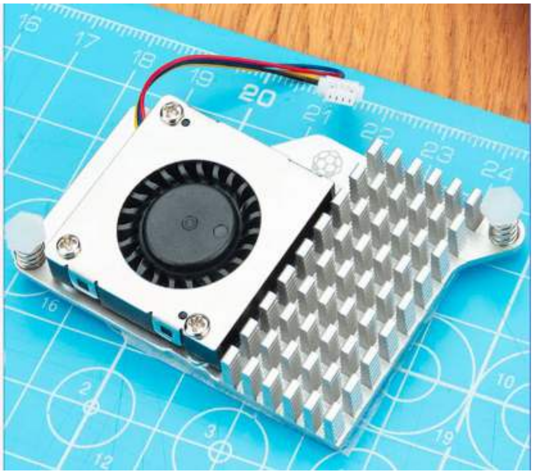
By the way, as Alasdair Allan discussed, inferencing directly on the Raspberry Pi 5 CPU—with no GPU acceleration—is now on par with the performance of the Coral TPU.



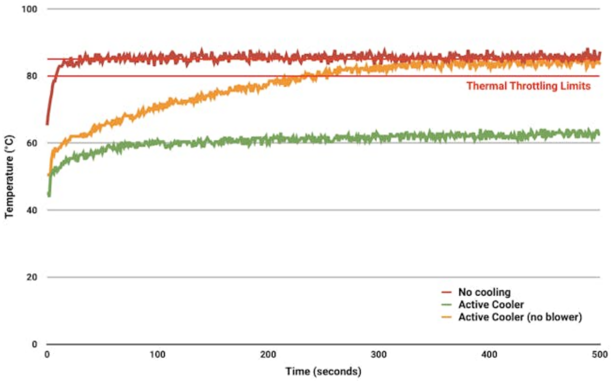
For more info, please see the complete article: Benchmarking TensorFlow and TensorFlow Lite on Raspberry Pi 5.

Raspberry Pi Active Cooler

We suggest installing an Active Cooler, a dedicated clip-on cooling solution for Raspberry Pi 5 (Raspi-5), for this lab. It combines an aluminum heatsink with a temperature-controlled blower fan to keep the Raspi-5 operating comfortably under heavy loads, such as running SLMs.



The Active Cooler has pre-applied thermal pads for heat transfer and is mounted directly to the Raspberry Pi 5 board using spring-loaded push pins. The Raspberry Pi firmware actively manages it: at 60°C, the blower's fan will be turned on; at 67.5°C, the fan speed will be increased; and finally, at 75°C, the fan increases to full speed. The blower's fan will spin down automatically when the temperature drops below these limits.



> To prevent overheating, all Raspberry Pi boards begin to throttle the processor when the temperature reaches 80°Cand throttle even further when it reaches the maximum temperature of 85°C (more detail here).

Generative AI (GenAI)

Generative AI is an artificial intelligence system capable of creating new, original content across various mediums such as **text, images, audio, and video**. These systems learn patterns from existing data and use that knowledge to generate novel outputs that didn't previously exist. **Large Language Models (LLMs)**, **Small Language Models (SLMs)**, and **multimodal models** can all be considered types of GenAI when used for generative tasks.

GenAI provides the conceptual framework for AI-driven content creation, with LLMs serving as powerful general-purpose text generators. SLMs adapt this technology for edge computing, while multimodal models extend GenAI capabilities across different data types. Together, they represent a spectrum of generative AI technologies, each with its strengths and applications, collectively driving AI-powered content creation and understanding.

Large Language Models (LLMs)

Large Language Models (LLMs) are advanced artificial intelligence systems that understand, process, and generate human-like text. These models are characterized by their massive scale in terms of the amount of data they are trained on and the number of parameters they contain. Critical aspects of LLMs include:

1. **Size**: LLMs typically contain billions of parameters. For example, GPT-3 has 175 billion parameters, while some newer models exceed a trillion parameters.
2. **Training Data**: They are trained on vast amounts of text data, often including books, websites, and other diverse sources, amounting to hundreds of gigabytes or even terabytes of text.
3. **Architecture**: Most LLMs use transformer-based architectures, which allow them to process and generate text by paying attention to different parts of the input simultaneously.
4. **Capabilities**: LLMs can perform a wide range of language tasks without specific fine-tuning, including:
   - Text generation
   - Translation
   - Summarization
   - Question answering
   - Code generation
   - Logical reasoning
5. **Few-shot Learning**: They can often understand and perform new tasks with minimal examples or instructions.

6. **Resource-Intensive**: Due to their size, LLMs typically require significant computational resources to run, often needing powerful GPUs or TPUs.
7. **Continual Development**: The field of LLMs is rapidly evolving, with new models and techniques constantly emerging.
8. **Ethical Considerations**: The use of LLMs raises important questions about bias, misinformation, and the environmental impact of training such large models.
9. **Applications**: LLMs are used in various fields, including content creation, customer service, research assistance, and software development.
10. **Limitations**: Despite their power, LLMs can produce incorrect or biased information and lack true understanding or reasoning capabilities.

We must note that we use large models beyond text, calling them *multi-modal models*. These models integrate and process information from multiple types of input simultaneously. They are designed to understand and generate content across various forms of data, such as text, images, audio, and video.

Closed vs Open Models:

Closed models, also called proprietary models, are AI models whose internal workings, code, and training data are not publicly disclosed. Examples: GPT-4 (by OpenAI), Claude (by Anthropic), Gemini (by Google).

Open models, also known as open-source models, are AI models whose underlying code, architecture, and often training data are publicly available and accessible. Examples: Gemma (by Google), LLaMA (by Meta) and Phi (by Microsoft).

Open models are particularly relevant for running models on edge devices like Raspberry Pi as they can be more easily adapted, optimized, and deployed in resource-constrained environments. Still, it is crucial to verify their Licenses. Open models come with various open-source licenses that may affect their use in commercial applications, while closed models have clear, albeit restrictive, terms of service.

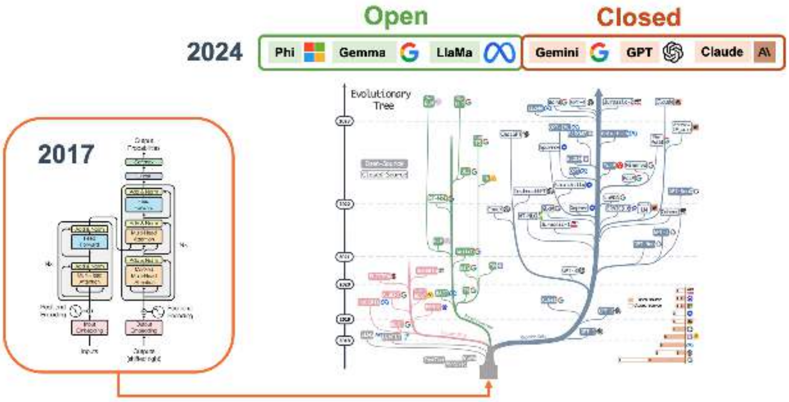


Figure 20.23: Adapted from arXiv

Small Language Models (SLMs)

In the context of edge computing on devices like Raspberry Pi, full-scale LLMs are typically too large and resource-intensive to run directly. This limitation has driven the development of smaller, more efficient models, such as the Small Language Models (SLMs).

SLMs are compact versions of LLMs designed to run efficiently on resource-constrained devices such as smartphones, IoT devices, and single-board computers like the Raspberry Pi. These models are significantly smaller in size and computational requirements than their larger counterparts while still retaining impressive language understanding and generation capabilities.

Key characteristics of SLMs include:

1. **Reduced parameter count**: Typically ranging from a few hundred million to a few billion parameters, compared to two-digit billions in larger models.
2. **Lower memory footprint**: Requiring, at most, a few gigabytes of memory rather than tens or hundreds of gigabytes.
3. **Faster inference time**: Can generate responses in milliseconds to seconds on edge devices.
4. **Energy efficiency**: Consuming less power, making them suitable for battery-powered devices.
5. **Privacy-preserving**: Enabling on-device processing without sending data to cloud servers.
6. **Offline functionality**: Operating without an internet connection.

SLMs achieve their compact size through various techniques such as knowledge distillation, model pruning, and quantization. While they may not match the broad capabilities of larger models, SLMs excel in specific tasks and domains, making them ideal for targeted applications on edge devices.

> We will generally consider SLMs, language models with less than 5 billion parameters quantized to 4 bits.

Examples of SLMs include compressed versions of models like Meta Llama, Microsoft PHI, and Google Gemma. These models enable a wide range of natural language processing tasks directly on edge devices, from text classification and sentiment analysis to question answering and limited text generation.

For more information on SLMs, the paper, LLM Pruning and Distillation in Practice: The Minitron Approach, provides an approach applying pruning and distillation to obtain SLMs from LLMs. And, SMALL LANGUAGE MODELS: SURVEY, MEASUREMENTS, AND INSIGHTS, presents a comprehensive survey and analysis of Small Language Models (SLMs), which are language models with 100 million to 5 billion parameters designed for resource-constrained devices.

Ollama

Figure 20.24: ollama logo

Ollama is an open-source framework that allows us to run language models (LMs), large or small, locally on our machines. Here are some critical points about Ollama:

1. **Local Model Execution**: Ollama enables running LMs on personal computers or edge devices such as the Raspi-5, eliminating the need for cloud-based API calls.
2. **Ease of Use**: It provides a simple command-line interface for downloading, running, and managing different language models.
3. **Model Variety**: Ollama supports various LLMs, including Phi, Gemma, Llama, Mistral, and other open-source models.
4. **Customization**: Users can create and share custom models tailored to specific needs or domains.
5. **Lightweight**: Designed to be efficient and run on consumer-grade hardware.
6. **API Integration**: Offers an API that allows integration with other applications and services.
7. **Privacy-Focused**: By running models locally, it addresses privacy concerns associated with sending data to external servers.

8. **Cross-Platform**: Available for macOS, Windows, and Linux systems (our case, here).
9. **Active Development**: Regularly updated with new features and model support.
10. **Community-Driven**: Benefits from community contributions and model sharing.

To learn more about what Ollama is and how it works under the hood, you should see this short video from Matt Williams, one of the founders of Ollama: https://www.youtube.com/embed/90ozfdsQOKo

> Matt has an entirely free course about Ollama that we recommend: https://youtu.be/9KEUFe4KQAI?si=D_-q3CMbHiT-twuy

Installing Ollama

Let's set up and activate a Virtual Environment for working with Ollama:

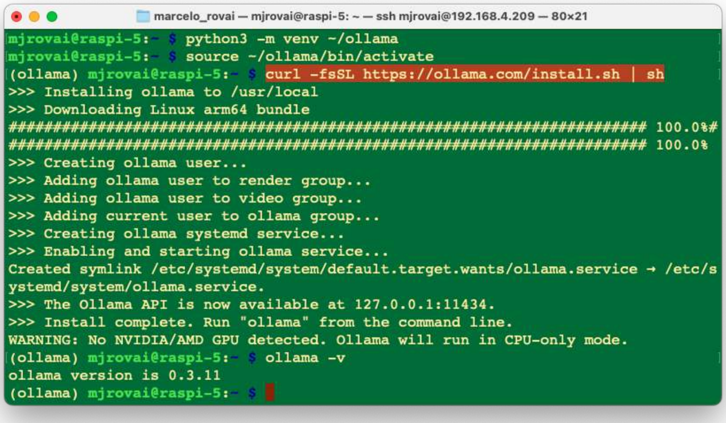
```
python3 -m venv ~/ollama
source ~/ollama/bin/activate
```

And run the command to install Ollama:

```
curl -fsSL https://ollama.com/install.sh | sh
```

As a result, an API will run in the background on `127.0.0.1:11434`. From now on, we can run Ollama via the terminal. For starting, let's verify the Ollama version, which will also tell us that it is correctly installed:

```
ollama -v
```


marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 80×21

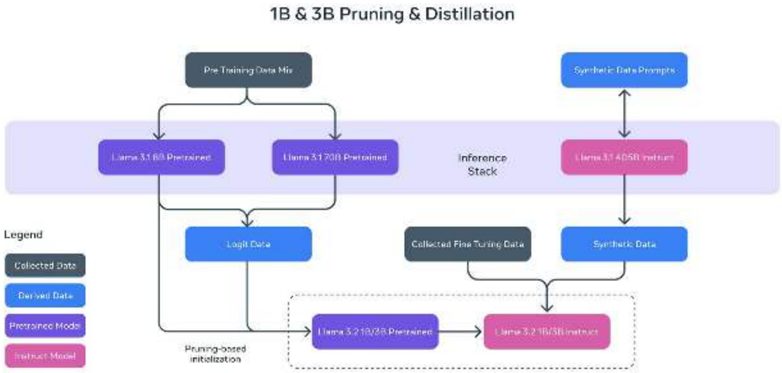
```
mjrovai@raspi-5:~ $ python3 -m venv ~/ollama
mjrovai@raspi-5:~ $ source ~/ollama/bin/activate
(ollama) mjrovai@raspi-5:~ $ curl -fsSL https://ollama.com/install.sh | sh
>>> Installing ollama to /usr/local
>>> Downloading Linux arm64 bundle
######################################################################## 100.0%#
######################################################################## 100.0%
>>> Creating ollama user...
>>> Adding ollama user to render group...
>>> Adding ollama user to video group...
>>> Adding current user to ollama group...
>>> Creating ollama systemd service...
>>> Enabling and starting ollama service...
Created symlink /etc/systemd/system/default.target.wants/ollama.service → /etc/s
ystemd/system/ollama.service.
>>> The Ollama API is now available at 127.0.0.1:11434.
>>> Install complete. Run "ollama" from the command line.
WARNING: No NVIDIA/AMD GPU detected. Ollama will run in CPU-only mode.
(ollama) mjrovai@raspi-5:~ $ ollama -v
ollama version is 0.3.11
(ollama) mjrovai@raspi-5:~ $
```

On the Ollama Library page, we can find the models Ollama supports. For example, by filtering by `Most popular`, we can see Meta Llama, Google Gemma, Microsoft Phi, LLaVa, etc.

Meta Llama 3.2 1B/3B

Let's install and run our first small language model, Llama 3.2 1B (and 3B). The Meta Llama 3.2 series comprises a set of multilingual generative language models available in 1 billion and 3 billion parameter sizes. These models are designed to process text input and generate text output. The instruction-tuned variants within this collection are specifically optimized for multilingual conversational applications, including tasks involving information retrieval and summarization with an agentic approach. When compared to many existing open-source and proprietary chat models, the Llama 3.2 instruction-tuned models demonstrate superior performance on widely-used industry benchmarks.

The 1B and 3B models were pruned from the Llama 8B, and then logits from the 8B and 70B models were used as token-level targets (token-level distillation). Knowledge distillation was used to recover performance (they were trained with 9 trillion tokens). The 1B model has 1,24B, quantized to integer (Q8_0), and the 3B, 3.12B parameters, with a Q4_0 quantization, which ends with a size of 1.3 GB and 2 GB, respectively. Its context window is 131,072 tokens.



Install and run the Model

```
ollama run llama3.2:1b
```

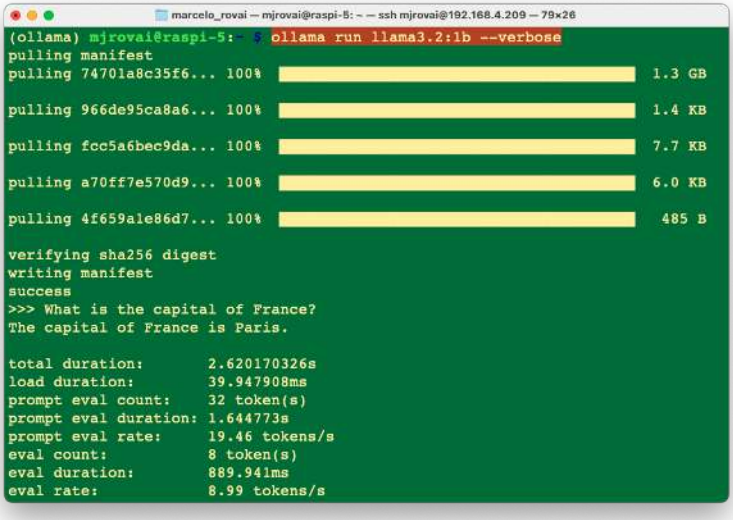
Running the model with the command before, we should have the Ollama prompt available for us to input a question and start chatting with the LLM model; for example,

`>>> What is the capital of France?`

Almost immediately, we get the correct answer:

`The capital of France is Paris.`

Using the option `--verbose` when calling the model will generate several statistics about its performance (The model will be polling only the first time we run the command).


marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 79×26

```
(ollama) mjrovai@raspi-5:~ $ ollama run llama3.2:1b --verbose
pulling manifest
pulling 74701a8c35f6... 100%                                        1.3 GB

pulling 966de95ca8a6... 100%                                        1.4 KB

pulling fcc5a6bec9da... 100%                                        7.7 KB

pulling a70ff7e570d9... 100%                                        6.0 KB

pulling 4f659a1e86d7... 100%                                         485 B

verifying sha256 digest
writing manifest
success
>>> What is the capital of France?
The capital of France is Paris.

total duration:       2.620170326s
load duration:        39.947908ms
prompt eval count:    32 token(s)
prompt eval duration: 1.644773s
prompt eval rate:     19.46 tokens/s
eval count:           8 token(s)
eval duration:        889.941ms
eval rate:            8.99 tokens/s
```

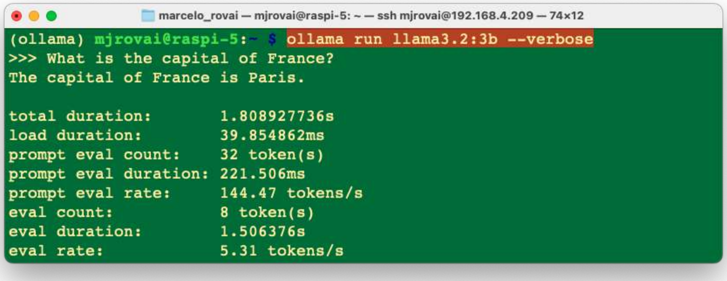
Each metric gives insights into how the model processes inputs and generates outputs. Here's a breakdown of what each metric means:

- **Total Duration (2.620170326 s)**: This is the complete time taken from the start of the command to the completion of the response. It encompasses loading the model, processing the input prompt, and generating the response.
- **Load Duration (39.947908 ms)**: This duration indicates the time to load the model or necessary components into memory. If this value is minimal, it can suggest that the model was preloaded or that only a minimal setup was required.
- **Prompt Eval Count (32 tokens)**: The number of tokens in the input prompt. In NLP, tokens are typically words or subwords, so this count includes all the tokens that the model evaluated to understand and respond to the query.

- **Prompt Eval Duration (1.644773 s)**: This measures the model's time to evaluate or process the input prompt. It accounts for the bulk of the total duration, implying that understanding the query and preparing a response is the most time-consuming part of the process.
- **Prompt Eval Rate (19.46 tokens/s)**: This rate indicates how quickly the model processes tokens from the input prompt. It reflects the model's speed in terms of natural language comprehension.
- **Eval Count (8 token(s))**: This is the number of tokens in the model's response, which in this case was, "The capital of France is Paris."
- **Eval Duration (889.941 ms)**: This is the time taken to generate the output based on the evaluated input. It's much shorter than the prompt evaluation, suggesting that generating the response is less complex or computationally intensive than understanding the prompt.
- **Eval Rate (8.99 tokens/s)**: Similar to the prompt eval rate, this indicates the speed at which the model generates output tokens. It's a crucial metric for understanding the model's efficiency in output generation.

This detailed breakdown can help understand the computational demands and performance characteristics of running SLMs like Llama on edge devices like the Raspberry Pi 5. It shows that while prompt evaluation is more time-consuming, the actual generation of responses is relatively quicker. This analysis is crucial for optimizing performance and diagnosing potential bottlenecks in real-time applications.

Loading and running the 3B model, we can see the difference in performance for the same prompt;


marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 74×12

```
(ollama) mjrovai@raspi-5:~ $ ollama run llama3.2:3b --verbose
>>> What is the capital of France?
The capital of France is Paris.

total duration:       1.808927736s
load duration:        39.854862ms
prompt eval count:    32 token(s)
prompt eval duration: 221.506ms
prompt eval rate:     144.47 tokens/s
eval count:           8 token(s)
eval duration:        1.506376s
eval rate:            5.31 tokens/s
```

The eval rate is lower, 5.3 tokens/s versus 9 tokens/s with the smaller model.

When question about

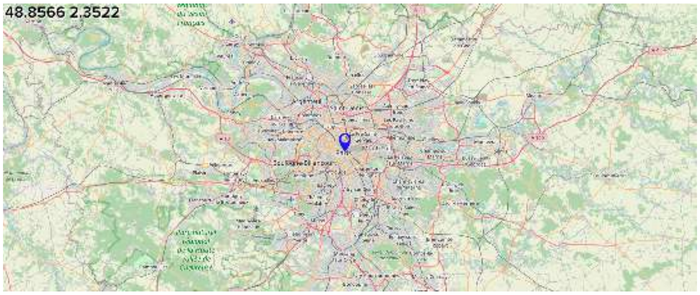
```
>>> What is the distance between Paris and Santiago, Chile?
```

The 1B model answered `9,841 kilometers (6,093 miles)`, which is inaccurate, and the 3B model answered `7,300 miles (11,700 km)`, which is close to the correct (11,642 km).

Let's ask for the Paris's coordinates:

```
>>> what is the latitude and longitude of Paris?
```

```
The latitude and longitude of Paris are 48.8567° N (48°55'
42" N) and 2.3510° E (2°22' 8" E), respectively.
```



Both 1B and 3B models gave correct answers.

Google Gemma 2 2B

Let's install Gemma 2, a high-performing and efficient model available in three sizes: 2B, 9B, and 27B. We will install **Gemma 2 2B**, a lightweight model trained with 2 trillion tokens that produces outsized results by learning from larger models through distillation. The model has 2.6 billion parameters and a Q4_0 quantization, which ends with a size of 1.6 GB. Its context window is 8,192 tokens.

Install and run the Model

```
ollama run gemma2:2b --verbose
```

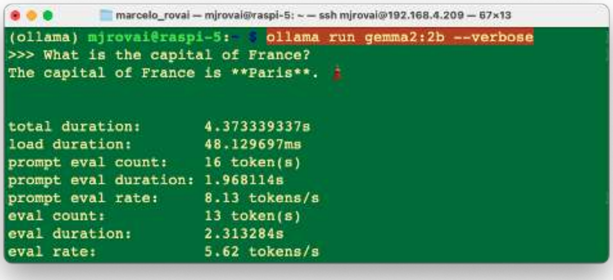
Running the model with the command before, we should have the Ollama prompt available for us to input a question and start chatting with the LLM model; for example,

```
>>> What is the capital of France?
```

Almost immediately, we get the correct answer:

```
The capital of France is **Paris**.
```

And it' statistics.


marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 67×13

```
(ollama) mjrovai@raspi-5:~ $ ollama run gemma2:2b --verbose
>>> What is the capital of France?
The capital of France is **Paris**.


total duration:       4.373339337s
load duration:        48.129697ms
prompt eval count:    16 token(s)
prompt eval duration: 1.968114s
prompt eval rate:     8.13 tokens/s
eval count:           13 token(s)
eval duration:        2.313284s
eval rate:            5.62 tokens/s
```

We can see that Gemma 2:2B has around the same performance as Llama 3.2:3B, but having less parameters.

Other examples:

```
>>> What is the distance between Paris and Santiago, Chile?

The distance between Paris, France and Santiago, Chile is
approximately **7,000 miles (11,267 kilometers)**.

Keep in mind that this is a straight-line distance, and actual
travel distance can vary depending on the chosen routes and any
stops along the way. `
```

Also, a good response but less accurate than Llama3.2:3B.

```
>>> what is the latitude and longitude of Paris?

You got it! Here are the latitudes and longitudes of Paris,
France:

* **Latitude**: 48.8566° N (north)
* **Longitude**: 2.3522° E (east)

Let me know if you'd like to explore more about Paris or its
location!
```

A good and accurate answer (a little more verbose than the Llama answers).

Microsoft Phi3.5 3.8B

Let's pull a bigger (but still tiny) model, the PHI3.5, a 3.8B lightweight state-of-the-art open model by Microsoft. The model belongs to the Phi-3 model family and supports `128K token` context length and the languages: Arabic, Chinese, Czech, Danish, Dutch, English, Finnish, French, German, Hebrew, Hungarian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Russian, Spanish, Swedish, Thai, Turkish and Ukrainian.

The model size, in terms of bytes, will depend on the specific quantization format used. The size can go from 2-bit quantization (q2_k) of 1.4 GB (higher performance/lower quality) to 16-bit quantization (fp-16) of 7.6 GB (lower performance/higher quality).

Let's run the 4-bit quantization (Q4_0), which will need 2.2 GB of RAM, with an intermediary trade-off regarding output quality and performance.

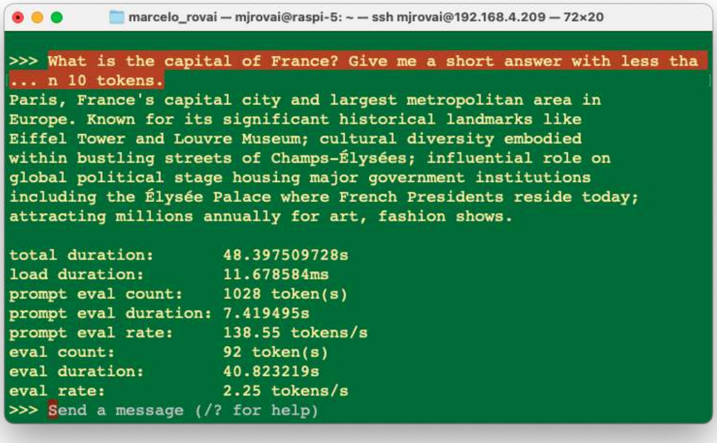
```
ollama run phi3.5:3.8b --verbose
```

> You can use run or pull to download the model. What happens is that Ollama keeps note of the pulled models, and once the PHI3 does not exist, before running it, Ollama pulls it.

Let's enter with the same prompt used before:

```
>>> What is the capital of France?

The capital of France is Paris. It' extradites significant
historical, cultural, and political importance to the country as
well as being a major European city known for its art, fashion,
gastronomy, and culture. Its influence extends beyond national
borders, with millions of tourists visiting each year from around
the globe. The Seine River flows through Paris before it reaches
the broader English Channel at Le Havre. Moreover, France is one
of Europe's leading economies with its capital playing a key role

...
```

The answer was very "verbose", let's specify a better prompt:


marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 72×20

```
>>> What is the capital of France? Give me a short answer with less tha
... n 10 tokens.
Paris, France's capital city and largest metropolitan area in
Europe. Known for its significant historical landmarks like
Eiffel Tower and Louvre Museum; cultural diversity embodied
within bustling streets of Champs-Élysées; influential role on
global political stage housing major government institutions
including the Élysée Palace where French Presidents reside today;
attracting millions annually for art, fashion shows.

total duration:       48.397509728s
load duration:        11.678584ms
prompt eval count:    1028 token(s)
prompt eval duration: 7.419495s
prompt eval rate:     138.55 tokens/s
eval count:           92 token(s)
eval duration:        40.823219s
eval rate:            2.25 tokens/s
>>> Send a message (/? for help)
```

In this case, the answer was still longer than we expected, with an eval rate of 2.25 tokens/s, more than double that of Gemma and Llama.

> Choosing the most appropriate prompt is one of the most important skills to be used with LLMs, no matter its size.

When we asked the same questions about distance and Latitude/Longitude, we did not get a good answer for a distance of `13,507 kilometers (8,429 miles)`, but it was OK for coordinates. Again, it could have been less verbose (more than 200 tokens for each answer).

We can use any model as an assistant since their speed is relatively decent, but on September 24 (2023), the Llama2:3B is a better choice. You should try other models, depending on your needs. Open LLM Leaderboard can give you an idea about the best models in size, benchmark, license, etc.

> The best model to use is the one fit for your specific necessity. Also, take into consideration that this field evolves with new models everyday.

Multimodal Models

Multimodal models are artificial intelligence (AI) systems that can process and understand information from multiple sources, such as images, text, audio, and video. In our context, multimodal LLMs can process various inputs, including text, images, and audio, as prompts and convert those prompts into various outputs, not just the source type.

We will work here with LLaVA-Phi-3, a fine-tuned LLaVA model from Phi 3 Mini 4k. It has strong performance benchmarks that are on par with the original LLaVA (Large Language and Vision Assistant) model.

The LLaVA-Phi-3 is an end-to-end trained large multimodal model designed to understand and generate content based on visual inputs (images) and textual instructions. It combines the capabilities of a visual encoder and a language model to process and respond to multimodal inputs.

Let's install the model:

```
ollama run llava-phi3:3.8b --verbose
```

Let's start with a text input:

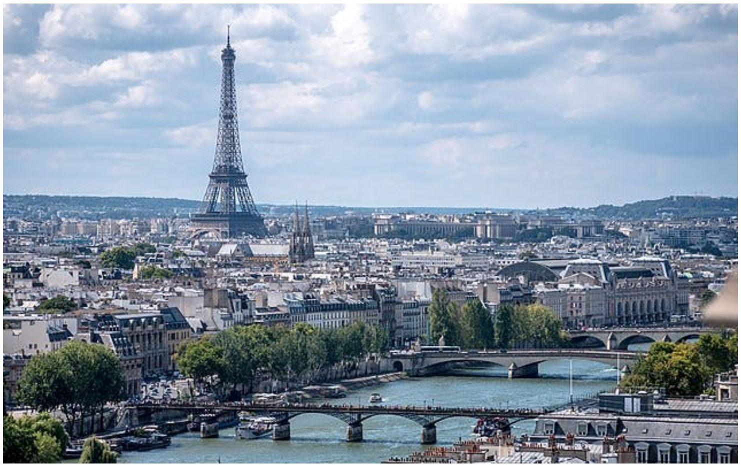
```
>>> You are a helpful AI assistant.
    What is the capital of France?

As an AI language model, I can tell you that the capital of
France is Paris. It's not only the largest city in the country
but also serves as its political and administrative center.
Paris is known for its iconic landmarks such as the Eiffel
Tower, Notre-Dame Cathedral, and the Louvre Museum. The city
has a rich history, beautiful architecture, and is widely
considered to be one of the most romantic cities in the world.
```

The response took around 30 s, with an eval rate of 3.93 tokens/s! Not bad!

But let us know to enter with an image as input. For that, let's create a directory for working:

```
cd Documents/
mkdir OLLAMA
cd OLLAMA
```

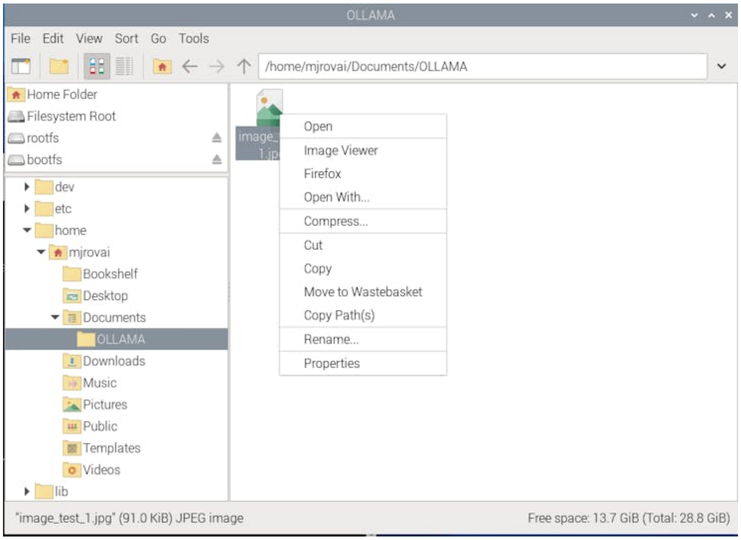
Let's download a 640 × 320 image from the internet, for example (Wikipedia: Paris, France):



Using FileZilla, for example, let's upload the image to the OLLAMA folder at the Raspi-5 and name it `image_test_1.jpg`. We should have the whole image path (we can use `pwd` to get it).

`/home/mjrovai/Documents/OLLAMA/image_test_1.jpg`

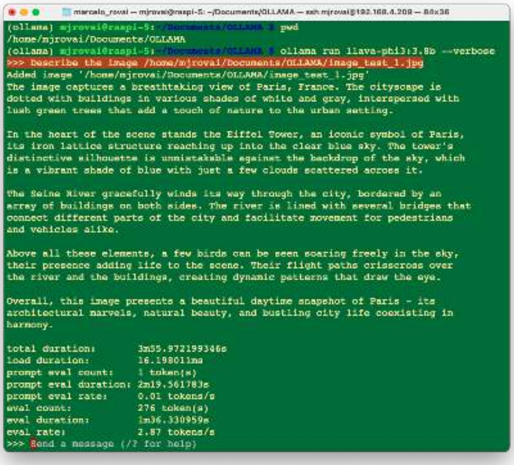
If you use a desktop, you can copy the image path by clicking the image with the mouse's right button.



Let's enter with this prompt:

```
>>> Describe the image /home/mjrovai/Documents/OLLAMA/\
                    image_test_1.jpg
```

The result was great, but the overall latency was significant; almost 4 minutes to perform the inference.


marcelo_rovai — mjrovai@raspi-5: ~/Documents/OLLAMA — ssh mjrovai@192.168.4.209 — 88x36

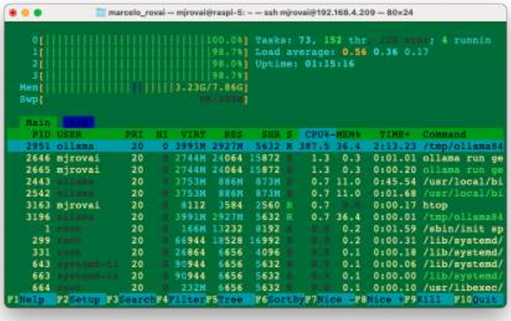
```
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ pwd
/home/mjrovai/Documents/OLLAMA
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ ollama run llava-phi3:3.8b --verbose
>>> Describe the image /home/mjrovai/Documents/OLLAMA/image_test_1.jpg
Added image '/home/mjrovai/Documents/OLLAMA/image_test_1.jpg'
The image captures a breathtaking view of Paris, France. The cityscape is
dotted with buildings in various shades of white and gray, interspersed with
lush green trees that add a touch of nature to the urban setting.

In the heart of the scene stands the Eiffel Tower, an iconic symbol of Paris,
its iron lattice structure reaching up into the clear blue sky. The tower's
distinctive silhouette is unmistakable against the backdrop of the sky, which
is a vibrant shade of blue with just a few clouds scattered across it.

The Seine River gracefully winds its way through the city, bordered by an
array of buildings on both sides. The river is lined with several bridges that
connect different parts of the city and facilitate movement for pedestrians
and vehicles alike.

Above all these elements, a few birds can be seen soaring freely in the sky,
their presence adding life to the scene. Their flight paths crisscross over
the river and the buildings, creating dynamic patterns that draw the eye.

Overall, this image presents a beautiful daytime snapshot of Paris - its
architectural marvels, natural beauty, and bustling city life coexisting in
harmony.

total duration:       3m55.972199346s
load duration:        16.198011ms
prompt eval count:    1 token(s)
prompt eval duration: 2m19.561783s
prompt eval rate:     0.01 tokens/s
eval count:           276 token(s)
eval duration:        1m36.330959s
eval rate:            2.87 tokens/s
>>> Send a message (/? for help)
```

Inspecting local resources

Using htop, we can monitor the resources running on our device.

```
htop
```

During the time that the model is running, we can inspect the resources:

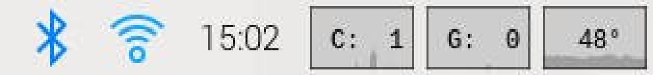


marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 80×24

```
    0[||||||||||||||||||||||||||||100.0%] Tasks: 73, 152 thr, [illegible] kthr; 4 runnin
    1[||||||||||||||||||||||||||||| 98.7%] Load average: 0.56 0.36 0.17
    2[||||||||||||||||||||||||||||| 98.0%] Uptime: 01:15:16
    3[||||||||||||||||||||||||||||| 98.7%]
  Mem[||||||||||||||||||||||3.23G/7.86G]
  Swp[                          0K/200M]

  Main  I/O
    PID USER       PRI  NI  VIRT   RES   SHR S  CPU%-MEM%   TIME+  Command
   2951 ollama      20   0 3991M 2927M  5632 R 387.5 36.4  2:13.23 /tmp/ollama84
   2646 mjrovai     20   0 2744M 24064 15872 S   1.3  0.3  0:01.01 ollama run ge
   2665 mjrovai     20   0 2744M 24064 15872 S   1.3  0.3  0:00.20 ollama run ge
   2443 ollama      20   0 3753M  886M  873M S   0.7 11.0  0:45.54 /usr/local/bi
   2542 ollama      20   0 3753M  886M  873M S   0.7 11.0  0:01.68 /usr/local/bi
   3163 mjrovai     20   0  8112  3584  2560 R   0.7  0.0  0:00.17 htop
   3196 ollama      20   0 3991M 2927M  5632 R   0.7 36.4  0:00.01 /tmp/ollama84
      1 root        20   0  166M 13232  8192 S   0.0  0.2  0:01.59 /sbin/init sp
    299 root        20   0 66944 18528 16992 S   0.0  0.2  0:00.31 /lib/systemd/
    331 root        20   0 26864  6656  4096 S   0.0  0.1  0:00.18 /lib/systemd/
    643 systemd-ti  20   0 90944  6656  5632 S   0.0  0.1  0:00.06 /lib/systemd/
    663 systemd-ti  20   0 90944  6656  5632 S   0.0  0.1  0:00.00 /lib/systemd/
    664 root        20   0  232M  6656  5632 S   0.0  0.1  0:00.10 /usr/libexec/
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

All four CPUs run at almost 100% of their capacity, and the memory used with the model loaded is `3.24 GB`. Exiting Ollama, the memory goes down to around `377 MB` (with no desktop).

It is also essential to monitor the temperature. When running the Raspberry with a desktop, you can have the temperature shown on the taskbar:



If you are "headless", the temperature can be monitored with the command:

```
vcgencmd measure_temp
```

If you are doing nothing, the temperature is around `50°C` for CPUs running at 1%. During inference, with the CPUs at 100%, the temperature can rise to almost `70°C`. This is OK and means the active cooler is working, keeping the temperature below 80°C / 85°C (its limit).

Ollama Python Library

So far, we have explored SLMs' chat capability using the command line on a terminal. However, we want to integrate those models into our projects, so Python seems to be the right path. The good news is that Ollama has such a library.

The Ollama Python library simplifies interaction with advanced LLM models, enabling more sophisticated responses and capabilities, besides providing the easiest way to integrate Python 3.8+ projects with Ollama.

For a better understanding of how to create apps using Ollama with Python, we can follow Matt Williams's videos, as the one below:

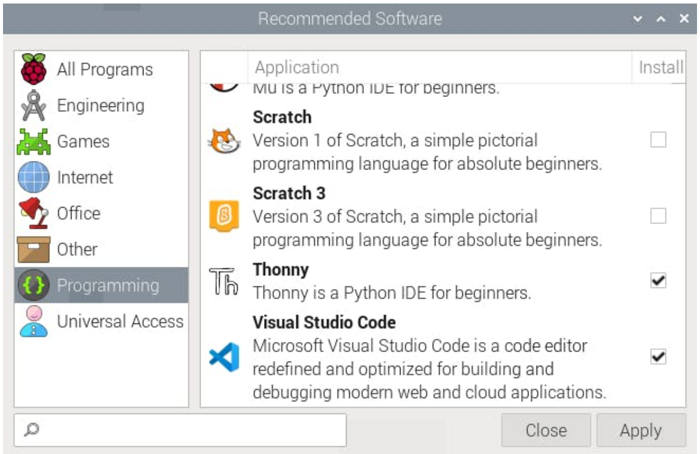
https://www.youtube.com/embed/_4K20tOsXK8

Installation:

In the terminal, run the command:

```
pip install ollama
```

We will need a text editor or an IDE to create a Python script. If you run the Raspberry OS on a desktop, several options, such as Thonny and Geany, have already been installed by default (accessed by `[Menu][Programming]`). You can download other IDEs, such as Visual Studio Code, from `[Menu][Recommended Software]`. When the window pops up, go to `[Programming]`, select the option of your choice, and press `[Apply]`.



Mu is a Python IDE for beginners.

Scratch ☐
Version 1 of Scratch, a simple pictorial programming language for absolute beginners.

Scratch 3 ☐
Version 3 of Scratch, a simple pictorial programming language for absolute beginners.

Thonny ☑
Thonny is a Python IDE for beginners.

Visual Studio Code ☑
Microsoft Visual Studio Code is a code editor redefined and optimized for building and debugging modern web and cloud applications.

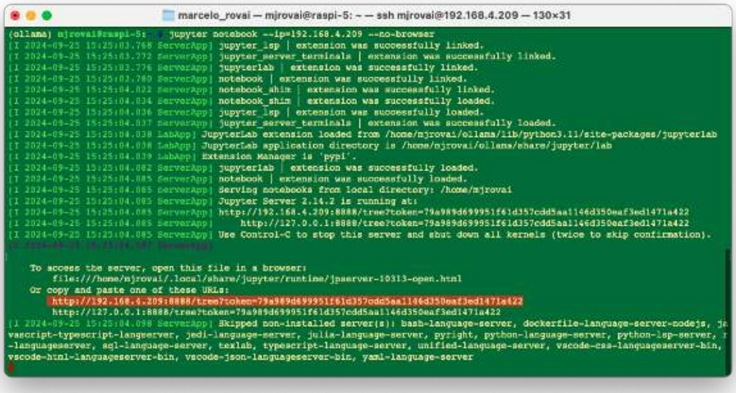
If you prefer using Jupyter Notebook for development:

```
pip install jupyter
jupyter notebook --generate-config
```

To run Jupyter Notebook, run the command (change the IP address for yours):

```
jupyter notebook --ip=192.168.4.209 --no-browser
```

On the terminal, you can see the local URL address to open the notebook:


marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 130×31

```
(ollama) mjrovai@raspi-5:~ $ jupyter notebook --ip=192.168.4.209 --no-browser
[I 2024-09-25 15:25:03.768 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2024-09-25 15:25:03.772 ServerApp] jupyter_server_terminals | extension was successfully linked.
[I 2024-09-25 15:25:03.776 ServerApp] jupyterlab | extension was successfully linked.
[I 2024-09-25 15:25:03.780 ServerApp] notebook | extension was successfully linked.
[I 2024-09-25 15:25:04.022 ServerApp] notebook_shim | extension was successfully linked.
[I 2024-09-25 15:25:04.034 ServerApp] notebook_shim | extension was successfully loaded.
[I 2024-09-25 15:25:04.036 ServerApp] jupyter_lsp | extension was successfully loaded.
[I 2024-09-25 15:25:04.037 ServerApp] jupyter_server_terminals | extension was successfully loaded.
[I 2024-09-25 15:25:04.038 LabApp] JupyterLab extension loaded from /home/mjrovai/ollama/lib/python3.11/site-packages/jupyterlab
[I 2024-09-25 15:25:04.038 LabApp] JupyterLab application directory is /home/mjrovai/ollama/share/jupyter/lab
[I 2024-09-25 15:25:04.039 LabApp] Extension Manager is 'pypi'.
[I 2024-09-25 15:25:04.082 ServerApp] jupyterlab | extension was successfully loaded.
[I 2024-09-25 15:25:04.085 ServerApp] notebook | extension was successfully loaded.
[I 2024-09-25 15:25:04.085 ServerApp] Serving notebooks from local directory: /home/mjrovai
[I 2024-09-25 15:25:04.085 ServerApp] Jupyter Server 2.14.2 is running at:
[I 2024-09-25 15:25:04.085 ServerApp] http://192.168.4.209:8888/tree?token=79a989d699951f61d357cdd5aa1146d350eaf3ed1471a422
[I 2024-09-25 15:25:04.085 ServerApp]     http://127.0.0.1:8888/tree?token=79a989d699951f61d357cdd5aa1146d350eaf3ed1471a422
[I 2024-09-25 15:25:04.085 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 2024-09-25 15:25:04.087 ServerApp]

    To access the server, open this file in a browser:
        file:///home/mjrovai/.local/share/jupyter/runtime/jpserver-10313-open.html
    Or copy and paste one of these URLs:
        http://192.168.4.209:8888/tree?token=79a989d699951f61d357cdd5aa1146d350eaf3ed1471a422
        http://127.0.0.1:8888/tree?token=79a989d699951f61d357cdd5aa1146d350eaf3ed1471a422
[I 2024-09-25 15:25:04.098 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfile-language-server-nodejs, ja
vascript-typescript-langserver, jedi-language-server, julia-language-server, pyright, python-language-server, python-lsp-server, r
-languageserver, sql-language-server, texlab, typescript-language-server, unified-language-server, vscode-css-languageserver-bin,
vscode-html-languageserver-bin, vscode-json-languageserver-bin, yaml-language-server
```

We can access it from another computer by entering the Raspberry Pi's IP address and the provided token in a web browser (we should copy it from the terminal).

In our working directory in the Raspi, we will create a new Python 3 notebook.

Let's enter with a very simple script to verify the installed models:

```python
import ollama
ollama.list()
```

All the models will be printed as a dictionary, for example:

```
  {'name': 'gemma2:2b',
   'model': 'gemma2:2b',
   'modified_at': '2024-09-24T19:30:40.053898094+01:00',
   'size': 1629518495,
   'digest': (
     '8ccf136fdd5298f3ffe2d69862750ea7fb56555fa4d5b18c0'
     '4e3fa4d82ee09d7'
    ),

   'details': {'parent_model': '',
    'format': 'gguf',
    'family': 'gemma2',
    'families': ['gemma2'],
    'parameter_size': '2.6B',
    'quantization_level': 'Q4_0'}}]}
```

Let's repeat one of the questions that we did before, but now using `ollama.generate()` from Ollama python library. This API will generate a response for the given prompt with the provided model. This is a streaming endpoint, so there will be a series of responses. The final response object will include statistics and additional data from the request.

```
MODEL = 'gemma2:2b'
PROMPT = 'What is the capital of France?'

res = ollama.generate(model=MODEL, prompt=PROMPT)
print (res)
```

In case you are running the code as a Python script, you should save it, for example, test_ollama.py. You can use the IDE to run it or do it directly on the terminal. Also, remember that you should always call the model and define it when running a stand-alone script.

```
python test_ollama.py
```

As a result, we will have the model response in a JSON format:

```
{
  'model': 'gemma2:2b',
  'created_at': '2024-09-25T14:43:31.869633807Z',
  'response': 'The capital of France is **Paris**.\n',
  'done': True,
  'done_reason': 'stop',
  'context': [
      106, 1645, 108, 1841, 603, 573, 6037, 576, 6081, 235336,
      107, 108, 106, 2516, 108, 651, 6037, 576, 6081, 603, 5231,
      29437, 168428, 235248, 244304, 241035, 235248, 108
  ],
  'total_duration': 24259469458,
  'load_duration': 19830013859,
  'prompt_eval_count': 16,
  'prompt_eval_duration': 1908757000,
  'eval_count': 14,
  'eval_duration': 2475410000
}
```

As we can see, several pieces of information are generated, such as:

- **response**: the main output text generated by the model in response to our prompt.
  - `The capital of France is **Paris**.`
- **context**: the token IDs representing the input and context used by the model. Tokens are numerical representations of text used for processing by the language model.
  - `[106, 1645, 108, 1841, 603, 573, 6037, 576, 6081, 235336, 107, 108, 106, 2516, 108, 651, 6037, 576, 6081, 603, 5231, 29437, 168428, 235248, 244304, 241035, 235248, 108]`

The Performance Metrics:

- **total_duration**: The total time taken for the operation in nanoseconds. In this case, approximately 24.26 seconds.
- **load_duration**: The time taken to load the model or components in nanoseconds. About 19.83 seconds.
- **prompt_eval_duration**: The time taken to evaluate the prompt in nanoseconds. Around 16 nanoseconds.
- **eval_count**: The number of tokens evaluated during the generation. Here, 14 tokens.
- **eval_duration**: The time taken for the model to generate the response in nanoseconds. Approximately 2.5 seconds.

But, what we want is the plain ‘response’ and, perhaps for analysis, the total duration of the inference, so let’s change the code to extract it from the dictionary:

```
print(f"\n{res['response']}")
print(
    f"\n [INFO] Total Duration: "
    f"{res['total_duration']/1e9:.2f} seconds"
)
```

Now, we got:

```
The capital of France is **Paris**.

 [INFO] Total Duration: 24.26 seconds
```

Using Ollama.chat()

Another way to get our response is to use `ollama.chat()`, which generates the next message in a chat with a provided model. This is a streaming endpoint, so a series of responses will occur. Streaming can be disabled using `"stream": false`. The final response object will also include statistics and additional data from the request.

```
PROMPT_1 = 'What is the capital of France?'

response = ollama.chat(model=MODEL, messages=[
{'role': 'user','content': PROMPT_1,},])
resp_1 = response['message']['content']
print(f"\n{resp_1}")
print(f"\n [INFO] Total Duration: "
      f"{(res['total_duration']/1e9):.2f} seconds")
```

The answer is the same as before.

An important consideration is that by using `ollama.generate()`, the response is “clear” from the model’s “memory” after the end of inference (only used once), but If we want to keep a conversation, we must use `ollama.chat()`. Let’s see it in action:

```
PROMPT_1 = 'What is the capital of France?'
response = ollama.chat(model=MODEL, messages=[
{'role': 'user','content': PROMPT_1,},])
resp_1 = response['message']['content']
print(f"\n{resp_1}")
print(f"\n [INFO] Total Duration: "
      f"{(response['total_duration']/1e9):.2f} seconds")


PROMPT_2 = 'and of Italy?'
response = ollama.chat(model=MODEL, messages=[
{'role': 'user','content': PROMPT_1,},
{'role': 'assistant','content': resp_1,},
{'role': 'user','content': PROMPT_2,},])
resp_2 = response['message']['content']
print(f"\n{resp_2}")
print(f"\n [INFO] Total Duration: "
      f"{(response_2['total_duration']/1e9):.2f} seconds")
```

In the above code, we are running two queries, and the second prompt considers the result of the first one.

Here is how the model responded:

```
The capital of France is **Paris**.

 [INFO] Total Duration: 2.82 seconds

The capital of Italy is **Rome**.

 [INFO] Total Duration: 4.46 seconds
```

Getting an image description:

In the same way that we have used the `LlaVa-PHI-3` model with the command line to analyze an image, the same can be done here with Python. Let's use the same image of Paris, but now with the `ollama.generate()`:

```
MODEL = 'llava-phi3:3.8b'
PROMPT = "Describe this picture"

with open('image_test_1.jpg', 'rb') as image_file:
    img = image_file.read()

response = ollama.generate(
    model=MODEL,
    prompt=PROMPT,
    images= [img]
)
```

```
print(f"\n{response['response']}")
print(f"\n [INFO] Total Duration: "
      f"{(res['total_duration']/1e9):.2f} seconds")
```

Here is the result:

```
This image captures the iconic cityscape of Paris, France. The
vantage point is high, providing a panoramic view of the Seine
River that meanders through the heart of the city. Several
bridges arch gracefully over the river, connecting different
parts of the city. The Eiffel Tower, an iron lattice structure
with a pointed top and two antennas on its summit, stands
tall in the background, piercing the sky. It is painted in a
light gray color, contrasting against the blue sky speckled
with white clouds.

The buildings that line the river are predominantly white or
beige, their uniform color palette broken occasionally by red
roofs peeking through. The Seine River itself appears calm
and wide, reflecting the city's architectural beauty in its
surface. On either side of the river, trees add a touch of
green to the urban landscape.

The image is taken from an elevated perspective, looking down
on the city. This viewpoint allows for a comprehensive view of
Paris's beautiful architecture and layout. The relative
positions of the buildings, bridges, and other structures
create a harmonious composition that showcases the city's charm.

In summary, this image presents a serene day in Paris, with its
architectural marvels - from the Eiffel Tower to the river-side
buildings - all bathed in soft colors under a clear sky.

 [INFO] Total Duration: 256.45 seconds
```

The model took about 4 minutes (256.45 s) to return with a detailed image description.

> In the 10-Ollama_Python_Library notebook, it is possible to find the experiments with the Ollama Python library.

Function Calling

So far, we can observe that by using the model's response into a variable, we can effectively incorporate it into real-world projects. However, a major issue arises when the model provides varying responses to the same input. For instance, let's assume that we only need the name of a country's capital and its coordinates as the model's response in the previous examples, without any additional information, even when utilizing verbose models like Microsoft Phi. To ensure consistent responses, we can employ the 'Ollama function call,' which is fully compatible with the OpenAI API.

But what exactly is "function calling"?

In modern artificial intelligence, function calling with Large Language Models (LLMs) allows these models to perform actions beyond generating text. By integrating with external functions or APIs, LLMs can access real-time data, automate tasks, and interact with various systems.

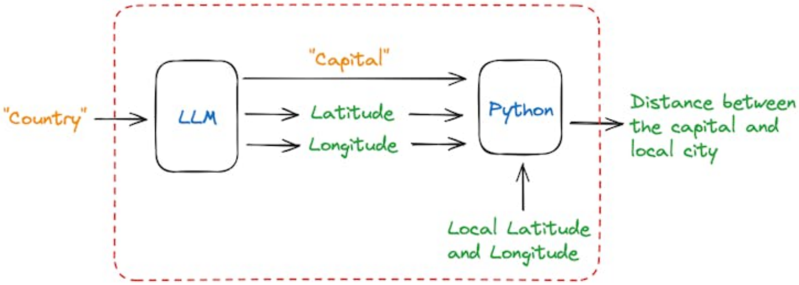
For instance, instead of merely responding to a query about the weather, an LLM can call a weather API to fetch the current conditions and provide accurate, up-to-date information. This capability enhances the relevance and accuracy of the model's responses and makes it a powerful tool for driving workflows and automating processes, transforming it into an active participant in real-world applications.

For more details about Function Calling, please see this video made by Marvin Prison:

https://www.youtube.com/embed/eHfMCtlsb1o

Let's create a project.

We want to create an *app* where the user enters a country's name and gets, as an output, the distance in km from the capital city of such a country and the app's location (for simplicity, We will use Santiago, Chile, as the app location).


"Country"
LLM
"Capital"
Latitude
Longitude
Python
Distance between the capital and local city
Local Latitude and Longitude

Once the user enters a country name, the model will return the name of its capital city (as a string) and the latitude and longitude of such city (in float). Using those coordinates, we can use a simple Python library (haversine) to calculate the distance between those 2 points.

The idea of this project is to demonstrate a combination of language model interaction, structured data handling with Pydantic, and geospatial calculations using the Haversine formula (traditional computing).

First, let us install some libraries. Besides *Haversine*, the main one is the OpenAI Python library, which provides convenient access to the OpenAI REST API from any Python 3.7+ application. The other one is Pydantic (and instructor), a robust data validation and settings management library engineered by Python to enhance the robustness and reliability of our codebase. In short, *Pydantic* will help ensure that our model's response will always be consistent.

```
pip install haversine
pip install openai
pip install pydantic
pip install instructor
```

Now, we should create a Python script designed to interact with our model (LLM) to determine the coordinates of a country's capital city and calculate the distance from Santiago de Chile to that capital.

Let's go over the code:

1. Importing Libraries

```
import sys
from haversine import haversine
from openai import OpenAI
from pydantic import BaseModel, Field
import instructor
```

- **sys**: Provides access to system-specific parameters and functions. It's used to get command-line arguments.
- **haversine**: A function from the haversine library that calculates the distance between two geographic points using the Haversine formula.
- **openAI**: A module for interacting with the OpenAI API (although it's used in conjunction with a local setup, Ollama). Everything is off-line here.
- **pydantic**: Provides data validation and settings management using Python-type annotations. It's used to define the structure of expected response data.
- **instructor**: A module is used to patch the OpenAI client to work in a specific mode (likely related to structured data handling).

2. Defining Input and Model

```
country = sys.argv[1]   # Get the country from
                        # command-line arguments
MODEL = 'phi3.5:3.8b'   # The name of the model to be used
mylat = -33.33          # Latitude of Santiago de Chile
mylon = -70.51          # Longitude of Santiago de Chile
```

- **country**: On a Python script, getting the country name from command-line arguments is possible. On a Jupyter notebook, we can enter its name, for example,
  - `country = "France"`
- **MODEL**: Specifies the model being used, which is, in this example, the phi3.5.

- **mylat and mylon**: Coordinates of Santiago de Chile, used as the starting point for the distance calculation.

3. Defining the Response Data Structure

```
class CityCoord(BaseModel):
    city: str = Field(
        ...,
        description="Name of the city"
    )
    lat: float = Field(
        ...,
        description="Decimal Latitude of the city"
    )
    lon: float = Field(
        ...,
        description="Decimal Longitude of the city"
    )
```

- **CityCoord**: A Pydantic model that defines the expected structure of the response from the LLM. It expects three fields: city (name of the city), lat (latitude), and lon (longitude).

4. Setting Up the OpenAI Client

```
client = instructor.patch(
    OpenAI(
        base_url="http://localhost:11434/v1",  # Local API base
                                               # URL (Ollama)
        api_key="ollama",                      # API key
                                               # (not used)
    ),
    mode=instructor.Mode.JSON,                 # Mode for
                                               # structured
                                               # JSON output
)
```

- **OpenAI**: This setup initializes an OpenAI client with a local base URL and an API key (ollama). It uses a local server.
- **instructor.patch**: Patches the OpenAI client to work in JSON mode, enabling structured output that matches the Pydantic model.

5. Generating the Response

```
resp = client.chat.completions.create(
    model=MODEL,
    messages=[
        {
            "role": "user",
            "content": f"return the decimal latitude and \
            decimal longitude of the capital of the {country}."
        }
    ],
    response_model=CityCoord,
    max_retries=10
)
```

- **client.chat.completions.create**: Calls the LLM to generate a response.
- **model**: Specifies the model to use (llava-phi3).
- **messages**: Contains the prompt for the LLM, asking for the latitude and longitude of the capital city of the specified country.
- **response_model**: Indicates that the response should conform to the CityCoord model.
- **max_retries**: The maximum number of retry attempts if the request fails.

6. Calculating the Distance

```
distance = haversine(
    (mylat, mylon),
    (resp.lat, resp.lon),
    unit='km'
)

print(
    f"Santiago de Chile is about {int(round(distance, -1))} "
    f"kilometers away from {resp.city}."
)
```

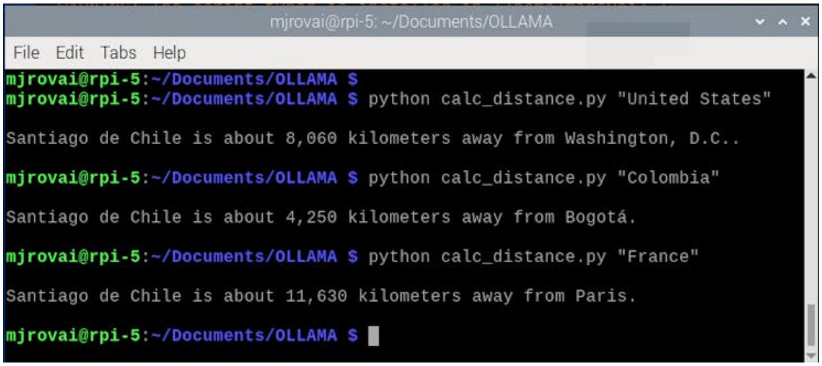
- **haversine**: Calculates the distance between Santiago de Chile and the capital city returned by the LLM using their respective coordinates.
- **(mylat, mylon)**: Coordinates of Santiago de Chile.
- **resp.city**: Name of the country's capital
- **(resp.lat, resp.lon)**: Coordinates of the capital city are provided by the LLM response.
- **unit = 'km'**: Specifies that the distance should be calculated in kilometers.
- **print**: Outputs the distance, rounded to the nearest 10 kilometers, with thousands of separators for readability.

Running the code

If we enter different countries, for example, France, Colombia, and the United States, We can note that we always receive the same structured information:

```
Santiago de Chile is about 8,060 kilometers away from
    Washington, D.C..
Santiago de Chile is about 4,250 kilometers away from Bogotá.
Santiago de Chile is about 11,630 kilometers away from Paris.
```

If you run the code as a script, the result will be printed on the terminal:



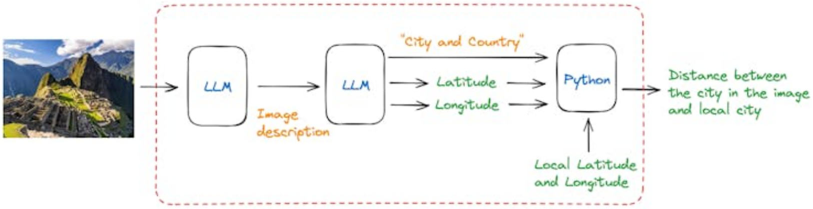
```
mjrovai@rpi-5:~/Documents/OLLAMA $
mjrovai@rpi-5:~/Documents/OLLAMA $ python calc_distance.py "United States"

Santiago de Chile is about 8,060 kilometers away from Washington, D.C..

mjrovai@rpi-5:~/Documents/OLLAMA $ python calc_distance.py "Colombia"

Santiago de Chile is about 4,250 kilometers away from Bogotá.

mjrovai@rpi-5:~/Documents/OLLAMA $ python calc_distance.py "France"

Santiago de Chile is about 11,630 kilometers away from Paris.

mjrovai@rpi-5:~/Documents/OLLAMA $ 
```

And the calculations are pretty good!

In the 20-Ollama_Function_Calling notebook, it is possible to find experiments with all models installed.

Adding images

Now it is time to wrap up everything so far! Let's modify the script so that instead of entering the country name (as a text), the user enters an image, and the application (based on SLM) returns the city in the image and its geographic location. With those data, we can calculate the distance as before.


LLM
Image description
LLM
"City and Country"
Latitude
Longitude
Python
Local Latitude and Longitude
Distance between the city in the image and local city

For simplicity, we will implement this new code in two steps. First, the LLM will analyze the image and create a description (text). This text will be passed on to another instance, where the model will extract the information needed to pass along.

We will start importing the libraries

```
import sys
import time
from haversine import haversine
import ollama
from openai import OpenAI
from pydantic import BaseModel, Field
import instructor
```

We can see the image if you run the code on the Jupyter Notebook. For that we need also import:

```
import matplotlib.pyplot as plt
from PIL import Image
```

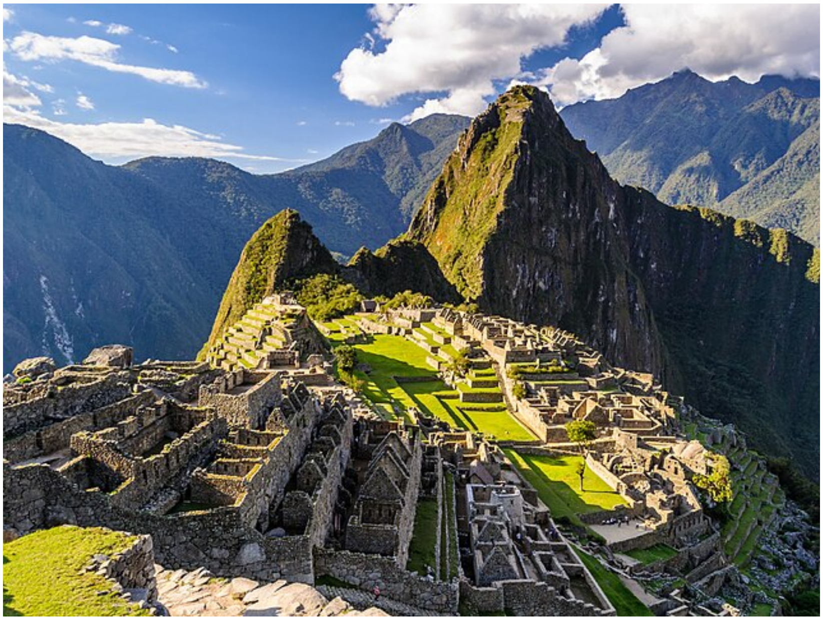
Those libraries are unnecessary if we run the code as a script.

Now, we define the model and the local coordinates:

```
MODEL = 'llava-phi3:3.8b'
mylat = -33.33
mylon = -70.51
```

We can download a new image, for example, Machu Picchu from Wikipedia. On the Notebook we can see it:

```
# Load the image
img_path = "image_test_3.jpg"
img = Image.open(img_path)

# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(img)
plt.axis('off')
#plt.title("Image")
plt.show()
```



Now, let's define a function that will receive the image and will `return the decimal latitude and decimal longitude of the city in the image, its name, and what country it is located`

```
def image_description(img_path):
    with open(img_path, 'rb') as file:
        response = ollama.chat(
            model=MODEL,
            messages=[
              {
                'role': 'user',
                'content': '''return the decimal latitude and \
                 decimal longitude of the city in the image, \
                 its name, and what country it is located''',
                'images': [file.read()],
              },
            ],
            options = {
              'temperature': 0,
              }
      )
    #print(response['message']['content'])
    return response['message']['content']
```

We can print the entire response for debug purposes.

The image description generated for the function will be passed as a prompt for the model again.

```
start_time = time.perf_counter()  # Start timing

class CityCoord(BaseModel):
    city: str = Field(
        ...,
        description="Name of the city in the image"
    )
    country: str = Field(
        ...,
        description=(
            "Name of the country where "
            "the city in the image is located"
        )
    )
    lat: float = Field(
        ...,
        description=(
            "Decimal latitude of the city in "
            "the image"
        )
    )
    lon: float = Field(
        ...,
        description=(
            "Decimal longitude of the city in "
            "the image"
        )
    )


# enables `response_model` in create call
client = instructor.patch(
    OpenAI(
        base_url="http://localhost:11434/v1",
        api_key="ollama"
    ),
    mode=instructor.Mode.JSON,
)

image_description = image_description(img_path)
# Send this description to the model
resp = client.chat.completions.create(
    model=MODEL,
    messages=[
        {
            "role": "user",
            "content": image_description,
        }
    ],
    response_model=CityCoord,
    max_retries=10,
    temperature=0,
)
```

If we print the image description , we will get:

```
The image shows the ancient city of Machu Picchu, located in
Peru. The city is perched on a steep hillside and consists of
various structures made of stone. It is surrounded by lush
greenery and towering mountains. The sky above is blue with
scattered clouds.

Machu Picchu's latitude is approximately 13.5086° S, and its
longitude is around 72.5494° W.
```

And the second response from the model (`resp`) will be:

```
CityCoord(city='Machu Picchu', country='Peru', lat=-13.5086,
                lon=-72.5494)
```

Now, we can do a "Post-Processing", calculating the distance and preparing the final answer:

```
distance = haversine(
    (mylat, mylon),
    (resp.lat, resp.lon),
    unit='km'
)

print((
    f"\nThe image shows {resp.city}, with lat: "
    f"{round(resp.lat, 2)} and long: "
    f"{round(resp.lon, 2)}, located in "
    f"{resp.country} and about "
    f"{int(round(distance, -1)):,} kilometers "
    f"away from Santiago, Chile.\n"
))


end_time = time.perf_counter()  # End timing
elapsed_time = end_time - start_time  # Calculate elapsed time
```

```
print(
    f"[INFO] ==> The code (running {MODEL}), "
    f"took {elapsed_time:.1f} seconds to execute.\n"
)
```

And we will get:

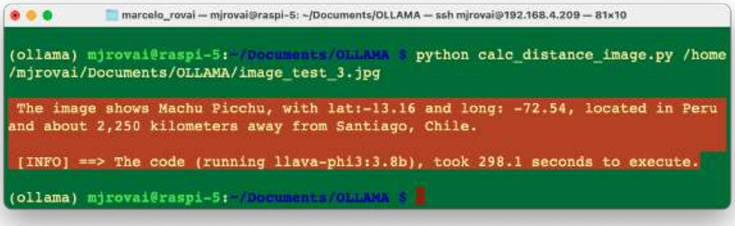
```
 The image shows Machu Picchu, with lat:-13.16 and long:
 -72.54, located in Peru  and about 2,250 kilometers away
 from Santiago, Chile.

print(
    f"[INFO] ==> The code (running {MODEL}), "
    f"took {elapsed_time:.1f} seconds "
    f"to execute.\n"
)
```

In the 30-Function_Calling_with_images notebook, it is possible to find the experiments with multiple images.

Let's now download the script `calc_distance_image.py` from the GitHub and run it on the terminal with the command:

```
python calc_distance_image.py \
  /home/mjrovai/Documents/OLLAMA/image_test_3.jpg
```

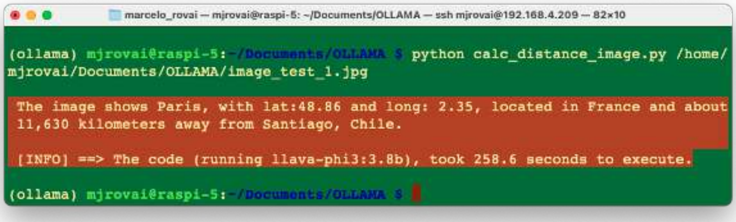
Enter with the Machu Picchu image full patch as an argument. We will get the same previous result.



marcelo_rovai — mjrovai@raspi-5: ~/Documents/OLLAMA — ssh mjrovai@192.168.4.209 — 81×10

```
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ python calc_distance_image.py /home
/mjrovai/Documents/OLLAMA/image_test_3.jpg

 The image shows Machu Picchu, with lat:-13.16 and long: -72.54, located in Peru
and about 2,250 kilometers away from Santiago, Chile.

 [INFO] ==> The code (running llava-phi3:3.8b), took 298.1 seconds to execute.

(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $
```

How about Paris?



marcelo_rovai — mjrovai@raspi-5: ~/Documents/OLLAMA — ssh mjrovai@192.168.4.209 — 82×10

```
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ python calc_distance_image.py /home/
mjrovai/Documents/OLLAMA/image_test_1.jpg

 The image shows Paris, with lat:48.86 and long: 2.35, located in France and about
 11,630 kilometers away from Santiago, Chile.

 [INFO] ==> The code (running llava-phi3:3.8b), took 258.6 seconds to execute.

(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $
```

Of course, there are many ways to optimize the code used here. Still, the idea is to explore the considerable potential of *function calling* with SLMs at the edge, allowing those models to integrate with external functions or APIs. Going beyond text generation, SLMs can access real-time data, automate tasks, and interact with various systems.

SLMs: Optimization Techniques

Large Language Models (LLMs) have revolutionized natural language processing, but their deployment and optimization come with unique challenges. One significant issue is the tendency for LLMs (and more, the SLMs) to generate plausible-sounding but factually incorrect information, a phenomenon known as **hallucination**. This occurs when models produce content that seems coherent but is not grounded in truth or real-world facts.

Other challenges include the immense computational resources required for training and running these models, the difficulty in maintaining up-to-date knowledge within the model, and the need for domain-specific adaptations. Privacy concerns also arise when handling sensitive data during training or inference. Additionally, ensuring consistent performance across diverse tasks and maintaining ethical use of these powerful tools present ongoing challenges. Addressing these issues is crucial for the effective and responsible deployment of LLMs in real-world applications.

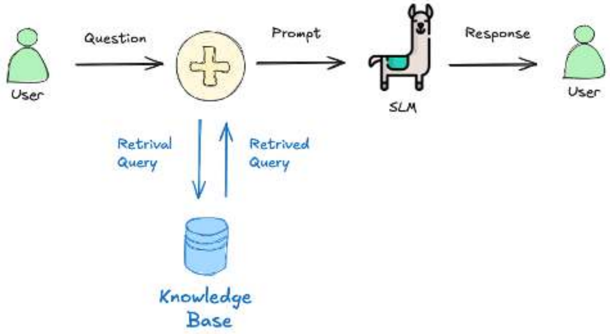
The fundamental techniques for enhancing LLM (and SLM) performance and efficiency are Fine-tuning, Prompt engineering, and Retrieval-Augmented Generation (RAG).

- **Fine-tuning**, while more resource-intensive, offers a way to specialize LLMs for particular domains or tasks. This process involves further training the model on carefully curated datasets, allowing it to adapt its vast general knowledge to specific applications. Fine-tuning can lead to substantial improvements in performance, especially in specialized fields or for unique use cases.
- **Prompt engineering** is at the forefront of LLM optimization. By carefully crafting input prompts, we can guide models to produce more accurate and relevant outputs. This technique involves structuring queries that leverage the model's pre-trained knowledge and capabilities, often incorporating examples or specific instructions to shape the desired response.
- **Retrieval-Augmented Generation (RAG)** represents another powerful approach to improving LLM performance. This method combines the vast knowledge embedded in pre-trained models with the ability to access and incorporate external, up-to-date information. By retrieving relevant data to supplement the model's decision-making process, RAG can significantly enhance accuracy and reduce the likelihood of generating outdated or false information.

For edge applications, it is more beneficial to focus on techniques like RAG that can enhance model performance without needing on-device fine-tuning. Let's explore it.

RAG Implementation

In a basic interaction between a user and a language model, the user asks a question, which is sent as a prompt to the model. The model generates a response based solely on its pre-trained knowledge. In a RAG process, there's an additional step between the user's question and the model's response. The user's question triggers a retrieval process from a knowledge base.



A simple RAG project

Here are the steps to implement a basic Retrieval Augmented Generation (RAG):

- **Determine the type of documents you'll be using**: The best types are documents from which we can get clean and unobscured text. PDFs can be problematic because they are designed for printing, not for extracting sensible text. To work with PDFs, we should get the source document or use tools to handle it.
- **Chunk the text**: We can't store the text as one long stream because of context size limitations and the potential for confusion. Chunking involves splitting the text into smaller pieces. Chunk text has many ways, such as character count, tokens, words, paragraphs, or sections. It is also possible to overlap chunks.
- **Create embeddings**: Embeddings are numerical representations of text that capture semantic meaning. We create embeddings by passing each chunk of text through a particular embedding model. The model outputs a vector, the length of which depends on the embedding model used. We should pull one (or more) embedding models from Ollama, to perform this task. Here are some examples of embedding models available at Ollama.

Model	Parameter Size	Embedding Size
mxbai-embed-large	334M	1024
nomic-embed-text	137M	768
all-minilm	23M	384

Generally, larger embedding sizes capture more nuanced information about the input. Still, they also require more computational resources to process, and a higher number of parameters should increase the latency (but also the quality of the response).

- **Store the chunks and embeddings in a vector database**: We will need a way to efficiently find the most relevant chunks of text for a given prompt, which is where a vector database comes in. We will use Chromadb, an AI-native open-source vector database, which simplifies building RAGs by creating knowledge, facts, and skills pluggable for LLMs. Both the embedding and the source text for each chunk are stored.
- **Build the prompt**: When we have a question, we create an embedding and query the vector database for the most similar chunks. Then, we select the top few results and include their text in the prompt.

The goal of RAG is to provide the model with the most relevant information from our documents, allowing it to generate more accurate and informative responses. So, let's implement a simple example of an SLM incorporating a particular set of facts about bees ("Bee Facts").

Inside the `ollama` env, enter the command in the terminal for Chromadb installation:

```
pip install ollama chromadb
```

Let's pull an intermediary embedding model, `nomic-embed-text`

```
ollama pull nomic-embed-text
```

And create a working directory:

```
cd Documents/OLLAMA/
mkdir RAG-simple-bee
cd RAG-simple-bee/
```

Let's create a new Jupyter notebook, 40-RAG-simple-bee for some exploration: Import the needed libraries:

```
import ollama
import chromadb
import time
```

And define aor models:

```
EMB_MODEL = "nomic-embed-text"
MODEL = 'llama3.2:3B'
```

Initially, a knowledge base about bee facts should be created. This involves collecting relevant documents and converting them into vector embeddings. These embeddings are then stored in a vector database, allowing for efficient similarity searches later. Enter with the "document," a base of "bee facts" as a list:

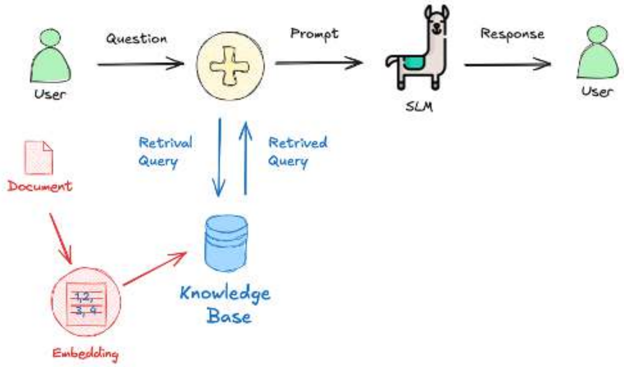
```
documents = [
    "Bee-keeping, also known as apiculture, involves the \
    maintenance of bee colonies, typically in hives, by humans.",
    "The most commonly kept species of bees is the European \
    honey bee (Apis  mellifera).",

    ...

    "There are another 20,000 different bee species in \
    the world.",
    "Brazil alone has more than 300 different bee species, and \
    the vast majority, unlike western honey bees, don't sting.",
    "Reports written in 1577 by Hans Staden, mention three \
    native bees used by indigenous people in Brazil.", \
    "The indigenous people in Brazil used bees for medicine \
    and food purposes",
    "From Hans Staden report: probable species: mandaçaia \
    (Melipona quadrifasciata), mandaguari (Scaptotrigona \
    postica) and jataí-amarela (Tetragonisca angustula)."
]
```

> We do not need to "chunk" the document here because we will use each element of the list and a chunk.

Now, we will create our vector embedding database `bee_facts` and store the document in it:

```
client = chromadb.Client()
collection = client.create_collection(name="bee_facts")

# store each document in a vector embedding database
for i, d in enumerate(documents):
  response = ollama.embeddings(model=EMB_MODEL, prompt=d)
  embedding = response["embedding"]
  collection.add(
    ids=[str(i)],
    embeddings=[embedding],
    documents=[d]
  )
```

Now that we have our "Knowledge Base" created, we can start making queries, retrieving data from it:



User Query: The process begins when a user asks a question, such as "How many bees are in a colony? Who lays eggs, and how much? How about common pests and diseases?"

```
prompt = "How many bees are in a colony? Who lays eggs and \
          how much? How about common pests and diseases?"
```

Query Embedding: The user's question is converted into a vector embedding using **the same embedding model** used for the knowledge base.

```
response = ollama.embeddings(
  prompt=prompt,
  model=EMB_MODEL
)
```

Relevant Document Retrieval: The system searches the knowledge base using the query embedding to find the most relevant documents (in this case, the 5 more probable). This is done using a similarity search, which compares the query embedding to the document embeddings in the database.

```
results = collection.query(
  query_embeddings=[response["embedding"]],
  n_results=5
)
data = results['documents']
```

Prompt Augmentation: The retrieved relevant information is combined with the original user query to create an augmented prompt. This prompt now contains the user's question and pertinent facts from the knowledge base.

```
prompt = (
    f"Using this data: {data}. "
    f"Respond to this prompt: {prompt}"
)
```

Answer Generation: The augmented prompt is then fed into a language model, in this case, the `llama3.2:3b` model. The model uses this enriched context to generate a comprehensive answer. Parameters like temperature, top_k, and top_p are set to control the randomness and quality of the generated response.

```
output = ollama.generate(
  model=MODEL,
  prompt = (
      f"Using this data: {data}. "
      f"Respond to this prompt: {prompt}"
  )

  options={
    "temperature": 0.0,
    "top_k":10,
    "top_p":0.5                          }
)
```

Response Delivery: Finally, the system returns the generated answer to the user.

```
print(output['response'])
```

```
Based on the provided data, here are the answers to your \
questions:

1. How many bees are in a colony?
A typical bee colony can contain between 20,000 and 80,000 bees.

2. Who lays eggs and how much?
The queen bee lays up to 2,000 eggs per day during peak seasons.

3. What about common pests and diseases?
Common pests and diseases that affect bees include varroa \
mites, hive beetles, and foulbrood.
```

Let's create a function to help answer new questions:

```
def rag_bees(prompt, n_results=5, temp=0.0, top_k=10, top_p=0.5):
    start_time = time.perf_counter()  # Start timing

    # generate an embedding for the prompt and retrieve the data
    response = ollama.embeddings(
      prompt=prompt,
      model=EMB_MODEL
    )
    results = collection.query(
      query_embeddings=[response["embedding"]],
      n_results=n_results
    )
    data = results['documents']

    # generate a response combining the prompt and data retrieved
    output = ollama.generate(
      model=MODEL,
      prompt = (
         f"Using this data: {data}. "
         f"Respond to this prompt: {prompt}"
      )

      options={
        "temperature": temp,
        "top_k": top_k,
        "top_p": top_p                          }
    )

    print(output['response'])

    end_time = time.perf_counter()  # End timing
    elapsed_time = round(
       (end_time - start_time), 1
    )  # Calculate elapsed time

print(
    f"\n[INFO] ==> The code for model: {MODEL}, "
    f"took {elapsed_time}s to generate the answer.\n"
)


    print(
       f"\n[INFO] ==> The code for model: {MODEL}, "
       f"took {elapsed_time}s to generate the answer.\n"
    )
```

We can now create queries and call the function:

```
prompt = "Are bees in Brazil?"
rag_bees(prompt)
```

```
Yes, bees are found in Brazil. According to the data, Brazil \
has more than 300 different bee species, and indigenous people \
in Brazil used bees for medicine and food purposes. \
Additionally, reports from 1577 mention three native bees \
used by indigenous people in Brazil.

 [INFO] ==> The code for model: llama3.2:3b, took 22.7s to \
 generate the answer.
```

By the way, if the model used supports multiple languages, we can use it (for example, Portuguese), even if the dataset was created in English:

```
prompt = "Existem abelhas no Brazil?"
rag_bees(prompt)
```

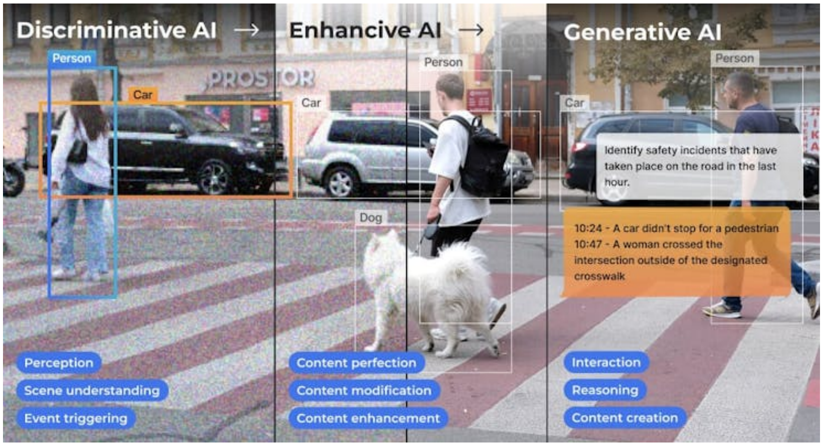
```
Sim, existem abelhas no Brasil! De acordo com o relato de Hans \
Staden, há três espécies de abelhas nativas do Brasil que \
foram mencionadas: mandaçaia (Melipona quadrifasciata), \
mandaguari (Scaptotrigona postica) e jataí-amarela \
(Tetragonisca angustula). Além disso, o Brasil é conhecido \
por ter mais de 300 espécies diferentes de abelhas, a \
maioria das quais não é agressiva e não põe veneno.

 [INFO] ==> The code for model: llama3.2:3b, took 54.6s to \
            generate the answer.
```

Going Further

The small LLM models tested worked well at the edge, both in text and with images, but of course, they had high latency regarding the last one. A combination of specific and dedicated models can lead to better results; for example, in real cases, an Object Detection model (such as YOLO) can get a general description and count of objects on an image that, once passed to an LLM, can help extract essential insights and actions.

According to Avi Baum, CTO at Hailo,

> In the vast landscape of artificial intelligence (AI), one of the most intriguing journeys has been the evolution of AI on the edge. This journey has taken us from classic machine vision to the realms of discriminative AI, enhancive AI, and now, the groundbreaking frontier of generative AI. Each step has brought us closer to a future where intelligent systems seamlessly integrate with our daily lives, offering an immersive experience of not just perception but also creation at the palm of our hand.



Conclusion

This lab has demonstrated how a Raspberry Pi 5 can be transformed into a potent AI hub capable of running large language models (LLMs) for real-time, on-site data analysis and insights using Ollama and Python. The Raspberry Pi's versatility and power, coupled with the capabilities of lightweight LLMs like Llama 3.2 and LLaVa-Phi-3-mini, make it an excellent platform for edge computing applications.

The potential of running LLMs on the edge extends far beyond simple data processing, as in this lab's examples. Here are some innovative suggestions for using this project:

1. **Smart Home Automation**:

   - Integrate SLMs to interpret voice commands or analyze sensor data for intelligent home automation. This could include real-time monitoring and control of home devices, security systems, and energy management, all processed locally without relying on cloud services.

2. **Field Data Collection and Analysis**:

   - Deploy SLMs on Raspberry Pi in remote or mobile setups for real-time data collection and analysis. This can be used in agriculture to monitor crop health, in environmental studies for wildlife tracking, or in disaster response for situational awareness and resource management.

3. **Educational Tools**:

   - Create interactive educational tools that leverage SLMs to provide instant feedback, language translation, and tutoring. This can be particularly useful in developing regions with limited access to advanced technology and internet connectivity.

4. **Healthcare Applications**:

   - Use SLMs for medical diagnostics and patient monitoring. They can provide real-time analysis of symptoms and suggest potential treatments.

This can be integrated into telemedicine platforms or portable health devices.

5. **Local Business Intelligence**:
   - Implement SLMs in retail or small business environments to analyze customer behavior, manage inventory, and optimize operations. The ability to process data locally ensures privacy and reduces dependency on external services.
6. **Industrial IoT**:
   - Integrate SLMs into industrial IoT systems for predictive maintenance, quality control, and process optimization. The Raspberry Pi can serve as a localized data processing unit, reducing latency and improving the reliability of automated systems.
7. **Autonomous Vehicles**:
   - Use SLMs to process sensory data from autonomous vehicles, enabling real-time decision-making and navigation. This can be applied to drones, robots, and self-driving cars for enhanced autonomy and safety.
8. **Cultural Heritage and Tourism**:
   - Implement SLMs to provide interactive and informative cultural heritage sites and museum guides. Visitors can use these systems to get real-time information and insights, enhancing their experience without internet connectivity.
9. **Artistic and Creative Projects**:
   - Use SLMs to analyze and generate creative content, such as music, art, and literature. This can foster innovative projects in the creative industries and allow for unique interactive experiences in exhibitions and performances.
10. **Customized Assistive Technologies**:
    - Develop assistive technologies for individuals with disabilities, providing personalized and adaptive support through real-time text-to-speech, language translation, and other accessible tools.

Resources

- 10-Ollama_Python_Library notebook
- 20-Ollama_Function_Calling notebook
- 30-Function_Calling_with_images notebook
- 40-RAG-simple-bee notebook
- calc_distance_image python script

Vision-Language Models (VLM)

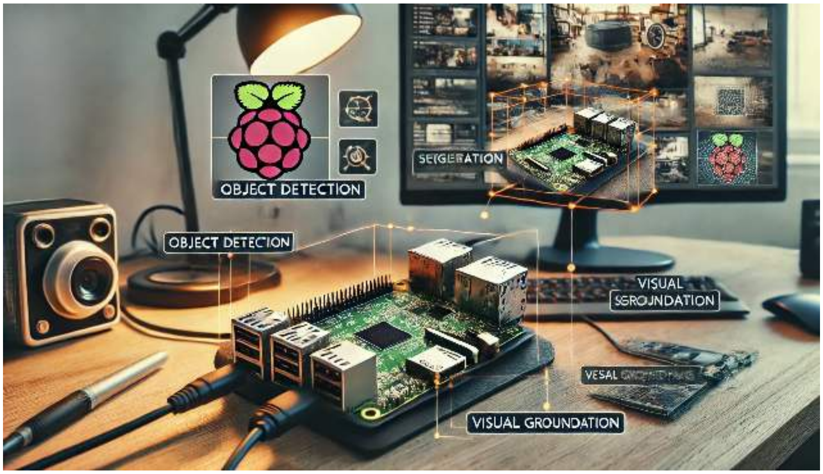


Figure 20.25: *DALL·E prompt - A Raspberry Pi setup featuring vision tasks. The image shows a Raspberry Pi connected to a camera, with various computer vision tasks displayed visually around it, including object detection, image captioning, segmentation, and visual grounding. The Raspberry Pi is placed on a desk, with a display showing bounding boxes and annotations related to these tasks. The background should be a home workspace, with tools and devices typically used by developers and hobbyists.*

Introduction

In this hands-on lab, we will continuously explore AI applications at the Edge, from the basic setup of the Florence-2, Microsoft's state-of-the-art vision foundation model, to advanced implementations on devices like the Raspberry Pi. We will learn to use Vision-Languageor Models (VLMs) for tasks such as captioning, object detection, grounding, segmentation, and OCR on a Raspberry Pi.

Why Florence-2 at the Edge?

Florence-2 is a vision-language model open-sourced by Microsoft under the MIT license, which significantly advances vision-language models by combining a lightweight architecture with robust capabilities. Thanks to its training on the massive FLD-5B dataset, which contains 126 million images and 5.4 billion visual annotations, it achieves performance comparable to larger models. This makes Florence-2 ideal for deployment at the edge, where power and computational resources are limited.

In this tutorial, we will explore how to use Florence-2 for real-time computer vision applications, such as:

- Image captioning
- Object detection
- Segmentation
- Visual grounding

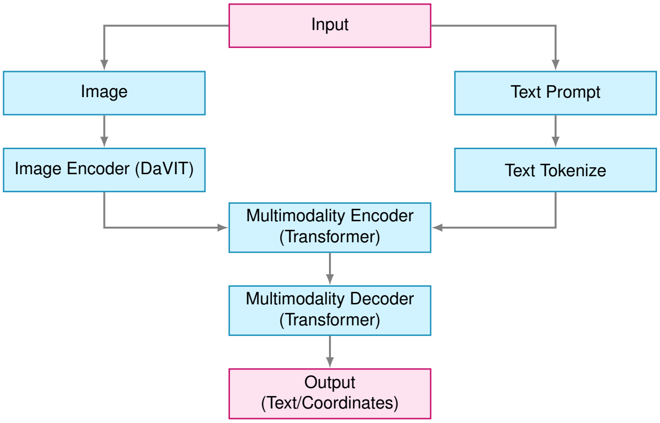
> **Visual grounding** involves linking textual descriptions to specific regions within an image. This enables the model to understand where particular objects or entities described in a prompt are in the image. For example, if the prompt is "a red car," the model will identify and highlight the region where the red car is found in the image. Visual grounding is helpful for applications where precise alignment between text and visual content is needed, such as human-computer interaction, image annotation, and interactive AI systems.

In the tutorial, we will walk through:

- Setting up Florence-2 on the Raspberry Pi
- Running inference tasks such as object detection and captioning
- Optimizing the model to get the best performance from the edge device
- Exploring practical, real-world applications with fine-tuning.

Florence-2 Model Architecture

Florence-2 utilizes a unified, prompt-based representation to handle various vision-language tasks. The model architecture consists of two main components: an **image encoder** and a **multi-modal transformer encoder-decoder**.


Input
Image
Text Prompt
Image Encoder (DaVIT)
Text Tokenize
Multimodality Encoder (Transformer)
Multimodality Decoder (Transformer)
Output (Text/Coordinates)

- **Image Encoder**: The image encoder is based on the DaViT (Dual Attention Vision Transformers) architecture. It converts input images into a series of visual token embeddings. These embeddings serve as the foundational representations of the visual content, capturing both spatial and contextual information about the image.

- **Multi-Modal Transformer Encoder-Decoder**: Florence-2's core is the multi-modal transformer encoder-decoder, which combines visual token embeddings from the image encoder with textual embeddings generated by a BERT-like model. This combination allows the model to simultaneously process visual and textual inputs, enabling a unified approach to tasks such as image captioning, object detection, and segmentation.

The model's training on the extensive FLD-5B dataset ensures it can effectively handle diverse vision tasks without requiring task-specific modifications. Florence-2 uses textual prompts to activate specific tasks, making it highly flexible and capable of zero-shot generalization. For tasks like object detection or visual grounding, the model incorporates additional location tokens to represent regions within the image, ensuring a precise understanding of spatial relationships.

> Florence-2's compact architecture and innovative training approach allow it to perform computer vision tasks accurately, even on resource-constrained devices like the Raspberry Pi.

Technical Overview

Florence-2 introduces several innovative features that set it apart:

Architecture

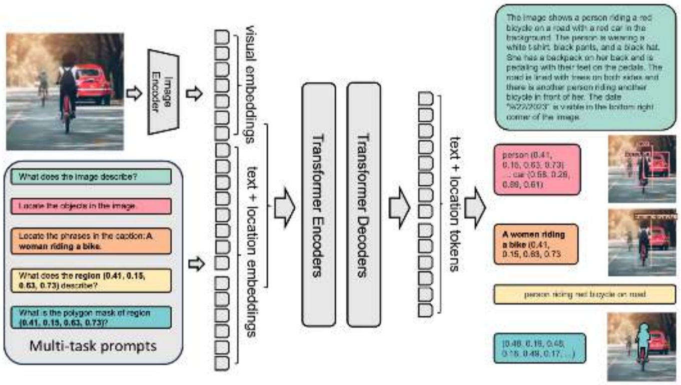
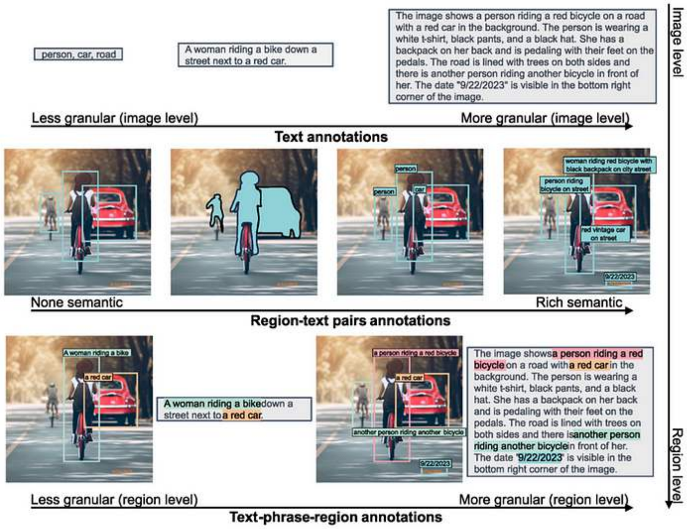


Image Encoder
visual embeddings
text + location embeddings
What does the image describe?
Locate the objects in the image.
Locate the phrases in the caption: A woman riding a bike.
What does the region (0.41, 0.15, 0.63, 0.73) describe?
What is the polygon mask of region (0.41, 0.15, 0.63, 0.73)?
Multi-task prompts
Transformer Encoders
Transformer Decoders
text + location tokens
The image shows a person riding a red bicycle on a road with a red car in the background. The person is wearing a white t-shirt, black pants, and a black hat. She has a backpack on her back and is pedaling with their feet on the pedals. The road is lined with trees on both sides and there is another person riding another bicycle in front of her. The date "9/22/2023" is visible in the bottom right corner of the image.
person (0.41, 0.15, 0.63, 0.73) ... car (0.58, 0.28, 0.89, 0.61)
A women riding a bike (0.41, 0.15, 0.63, 0.73
person riding red bicycle on road
(0.48, 0.15, 0.48, 0.15, 0.48, 0.17, ...)

- **Lightweight Design**: Two variants available
  - Florence-2-Base: 232 million parameters
  - Florence-2-Large: 771 million parameters
- **Unified Representation**: Handles multiple vision tasks through a single architecture
- **DaViT Vision Encoder**: Converts images into visual token embeddings
- **Transformer-based Multi-modal Encoder-Decoder**: Processes combined visual and text embeddings

Training Dataset (FLD-5B)



A woman riding a bike down a street next to a red car.

The image shows a person riding a red bicycle on a road with a red car in the background. The person is wearing a white t-shirt, black pants, and a black hat. She has a backpack on her back and is pedaling with their feet on the pedals. The road is lined with trees on both sides and there is another person riding another bicycle in front of her. The date "9/22/2023" is visible in the bottom right corner of the image.

A woman riding a bikedown a street next toa red car.

The image showsa person riding a red bicycle on a road witha red car in the background. The person is wearing a white t-shirt, black pants, and a black hat. She has a backpack on her back and is pedaling with their feet on the pedals. The road is lined with trees on both sides and there isanother person riding another bicyclein front of her. The date "9/22/2023" is visible in the bottom right corner of the image.

person, car, road
Image level
Less granular (image level)
More granular (image level)
Text annotations
person
person
car
woman riding red bicycle with black backpack on city street
person riding bicycle on street
red vintage car on street
9/22/2023
None semantic
Rich semantic
Region-text pairs annotations
A woman riding a bike
a red car
a person riding a red bicycle
a red car
another person riding another bicycle
9/22/2023
Less granular (region level)
More granular (region level)
Text-phrase-region annotations
Region level

- 126 million unique images
- 5.4 billion comprehensive annotations, including:
  - 500M text annotations
  - 1.3B region-text annotations
  - 3.6B text-phrase-region annotations
- Automated annotation pipeline using specialist models
- Iterative refinement process for high-quality labels

Key Capabilities

Florence-2 excels in multiple vision tasks:

Zero-shot Performance

- Image Captioning: Achieves 135.6 CIDEr score on COCO
- Visual Grounding: 84.4% recall@1 on Flickr30k
- Object Detection: 37.5 mAP on COCO val2017
- Referring Expression: 67.0% accuracy on RefCOCO

Fine-tuned Performance

- Competitive with specialist models despite the smaller size
- Outperforms larger models in specific benchmarks
- Efficient adaptation to new tasks

Practical Applications

Florence-2 can be applied across various domains:

1. **Content Understanding**
   - Automated image captioning for accessibility
   - Visual content moderation
   - Media asset management
2. **E-commerce**
   - Product image analysis
   - Visual search
   - Automated product tagging
3. **Healthcare**
   - Medical image analysis
   - Diagnostic assistance
   - Research data processing
4. **Security & Surveillance**
   - Object detection and tracking
   - Anomaly detection
   - Scene understanding

Comparing Florence-2 with other VLMs

Florence-2 stands out from other visual language models due to its impressive zero-shot capabilities. Unlike models like Google PaliGemma, which rely on extensive fine-tuning to adapt to various tasks, Florence-2 works right out of the box, as we will see in this lab. It can also compete with larger models like GPT-4V and Flamingo, which often have many more parameters but only sometimes match Florence-2's performance. For example, Florence-2 achieves better zero-shot results than Kosmos-2 despite having over twice the parameters.

In benchmark tests, Florence-2 has shown remarkable performance in tasks like COCO captioning and referring expression comprehension. It outperformed models like PolyFormer and UNINEXT in object detection and segmentation tasks on the COCO dataset. It is a highly competitive choice for real-world applications where both performance and resource efficiency are crucial.

Setup and Installation

Our choice of edge device is the Raspberry Pi 5 (Raspi-5). Its robust platform is equipped with the Broadcom BCM2712, a 2.4 GHz quad-core 64-bit Arm Cortex-A76 CPU featuring Cryptographic Extension and enhanced caching capabilities. It boasts a VideoCore VII GPU, dual 4Kp60 HDMI® outputs with HDR, and a 4Kp60 HEVC decoder. Memory options include 4 GB and 8 GB of high-speed LPDDR4X SDRAM, with 8 GB being our choice to run Florence-2. It also features expandable storage via a microSD card slot and a PCIe 2.0 interface for fast peripherals such as M.2 SSDs (Solid State Drives).

> For real applications, SSDs are a better option than SD cards.

We suggest installing an Active Cooler, a dedicated clip-on cooling solution for Raspberry Pi 5 (Raspi-5), for this lab. It combines an aluminum heatsink with a temperature-controlled blower fan to keep the Raspi-5 operating comfortably under heavy loads, such as running Florense-2.

Environment configuration

To run Microsoft Florense-2 on the Raspberry Pi 5, we'll need a few libraries:

1. **Transformers**:
   - Florence-2 uses the `transformers` library from Hugging Face for model loading and inference. This library provides the architecture for working with pre-trained vision-language models, making it easy to perform tasks like image captioning, object detection, and more. Essentially, `transformers` helps in interacting with the model, processing input prompts, and obtaining outputs.
2. **PyTorch**:
   - PyTorch is a deep learning framework that provides the infrastructure needed to run the Florence-2 model, which includes tensor operations, GPU acceleration (if a GPU is available), and model training/inference functionalities. The Florence-2 model is trained in PyTorch, and we need it to leverage its functions, layers, and computation capabilities to perform inferences on the Raspberry Pi.
3. **Timm** (PyTorch Image Models):
   - Florence-2 uses `timm` to access efficient implementations of vision models and pre-trained weights. Specifically, the `timm` library is utilized for the **image encoder** part of Florence-2, particularly for

    managing the DaViT architecture. It provides model definitions and optimized code for common vision tasks and allows the easy integration of different backbones that are lightweight and suitable for edge devices.

4. **Einops**:
   - `Einops` is a library for flexible and powerful tensor operations. It makes it easy to reshape and manipulate tensor dimensions, which is especially important for the multi-modal processing done in Florence-2. Vision-language models like Florence-2 often need to rearrange image data, text embeddings, and visual embeddings to align correctly for the transformer blocks, and `einops` simplifies these complex operations, making the code more readable and concise.

In short, these libraries enable different essential components of Florence-2:

- **Transformers** and **PyTorch** are needed to load the model and run the inference.
- **Timm** is used to access and efficiently implement the vision encoder.
- **Einops** helps reshape data, facilitating the integration of visual and text features.

All these components work together to help Florence-2 run seamlessly on our Raspberry Pi, allowing it to perform complex vision-language tasks relatively quickly.

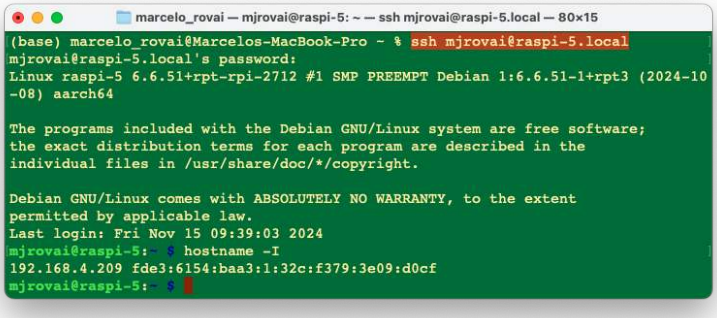
Considering that the Raspberry Pi already has its OS installed, let's use `SSH` to reach it from another computer:

```
ssh mjrovai@raspi-5.local
```

And check the IP allocated to it:

```
hostname -I
```

```
192.168.4.209
```



marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@raspi-5.local — 80×15

```
(base) marcelo_rovai@Marcelos-MacBook-Pro ~ % ssh mjrovai@raspi-5.local
mjrovai@raspi-5.local's password:
Linux raspi-5 6.6.51+rpt-rpi-2712 #1 SMP PREEMPT Debian 1:6.6.51-1+rpt3 (2024-10
-08) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Nov 15 09:39:03 2024
mjrovai@raspi-5:~ $ hostname -I
192.168.4.209 fde3:6154:baa3:1:32c:f379:3e09:d0cf
mjrovai@raspi-5:~ $
```

Updating the Raspberry Pi

First, ensure your Raspberry Pi is up to date:

```
sudo apt update
sudo apt upgrade -y
```

Initial setup for using PIP:

```
sudo apt install python3-pip
sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED
pip3 install --upgrade pip
```

Install Dependencies

```
sudo apt-get install libjpeg-dev libopenblas-dev libopenmpi-dev \
    libomp-dev
```

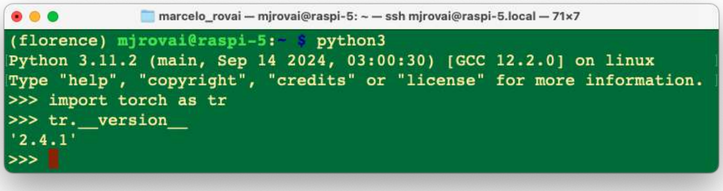
Let's set up and activate a **Virtual Environment** for working with Florence-2:

```
python3 -m venv ~/florence
source ~/florence/bin/activate
```

Install PyTorch

```
pip3 install setuptools numpy Cython
pip3 install requests
pip3 install torch torchvision \
    --index-url https://download.pytorch.org/whl/cpu
pip3 install torchaudio \
    --index-url https://download.pytorch.org/whl/cpu
```

Let's verify that PyTorch is correctly installed:



marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@raspi-5.local — 71×7

```
(florence) mjrovai@raspi-5:~ $ python3
Python 3.11.2 (main, Sep 14 2024, 03:00:30) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch as tr
>>> tr.__version__
'2.4.1'
>>>
```

Install Transformers, Timm and Einops:

```
pip3 install transformers
pip3 install timm einops
```

Install the model:

```
pip3 install autodistill-florence-2
```

Jupyter Notebook and Python libraries
Installing a Jupyter Notebook to run and test our Python scripts is possible.

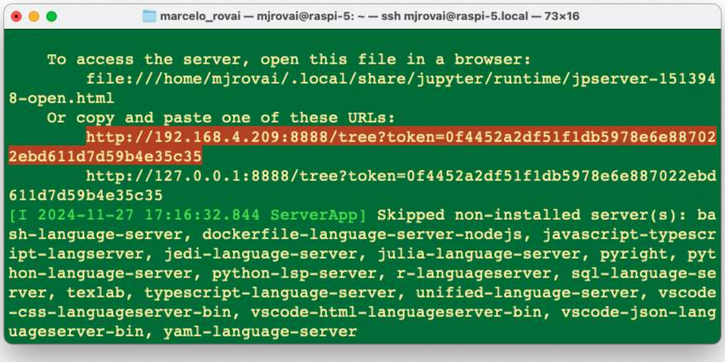
```
pip3 install jupyter
pip3 install numpy Pillow matplotlib
jupyter notebook --generate-config
```

Testing the installation

Running the Jupyter Notebook on the remote computer

```
jupyter notebook --ip=192.168.4.209 --no-browser
```

Running the above command on the SSH terminal, we can see the local URL address to open the notebook:


marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@raspi-5.local — 73×16

```
    To access the server, open this file in a browser:
        file:///home/mjrovai/.local/share/jupyter/runtime/jpserver-151394
8-open.html
    Or copy and paste one of these URLs:
        http://192.168.4.209:8888/tree?token=0f4452a2df51f1db5978e6e88702
2ebd611d7d59b4e35c35
        http://127.0.0.1:8888/tree?token=0f4452a2df51f1db5978e6e887022ebd
611d7d59b4e35c35
[I 2024-11-27 17:16:32.844 ServerApp] Skipped non-installed server(s): ba
sh-language-server, dockerfile-language-server-nodejs, javascript-typescr
ipt-langserver, jedi-language-server, julia-language-server, pyright, pyt
hon-language-server, python-lsp-server, r-languageserver, sql-language-se
rver, texlab, typescript-language-server, unified-language-server, vscode
-css-languageserver-bin, vscode-html-languageserver-bin, vscode-json-lang
uageserver-bin, yaml-language-server
```

The notebook with the code used on this initial test can be found on the Lab GitHub:

- 10-florence2_test.ipynb

We can access it on the remote computer by entering the Raspberry Pi's IP address and the provided token in a web browser (copy the entire URL from the terminal).

From the Home page, create a new notebook [`Python 3 (ipykernel)`] and copy and paste the example code from Hugging Face Hub.

The code is designed to run Florence-2 on a given image to perform **object detection**. It loads the model, processes an image and a prompt, and then generates a response to identify and describe the objects in the image.

- The **processor** helps prepare text and image inputs.
- The **model** takes the processed inputs to generate a meaningful response.

- The **post-processing** step refines the generated output into a more interpretable form, like bounding boxes for detected objects.

  This workflow leverages the versatility of Florence-2 to handle **vision-language tasks** and is implemented efficiently using PyTorch, Transformers, and related image-processing tools.

```
import requests
from PIL import Image
import torch
from transformers import AutoProcessor, AutoModelForCausalLM

device = "cuda:0" if torch.cuda.is_available() else "cpu"
torch_dtype = (
   torch.float16 if torch.cuda.is_available() else torch.float32
)

model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Florence-2-base",
    torch_dtype=torch_dtype,
    trust_remote_code=True
).to(device)
processor = AutoProcessor.from_pretrained(
    "microsoft/Florence-2-base",
    trust_remote_code=True
)

prompt = "<OD>"

url = (
    "https://huggingface.co/datasets/huggingface/"
    "documentation-images/resolve/main/transformers/"
    "tasks/car.jpg?download=true"
)
image = Image.open(requests.get(url, stream=True).raw)

inputs = processor(
    text=prompt,
    images=image,
    return_tensors="pt"
).to(device, torch_dtype)

generated_ids = model.generate(
    input_ids=inputs["input_ids"],
    pixel_values=inputs["pixel_values"],
    max_new_tokens=1024,
    do_sample=False,
    num_beams=3,
)
generated_text = processor.batch_decode(
   generated_ids, skip_special_tokens=False)[0]

parsed_answer = processor.post_process_generation(
    generated_text,
    task="<OD>",
    image_size=(image.width, image.height)
)

print(parsed_answer)
```

Let's break down the provided code step by step:

Importing Required Libraries

```
import requests
from PIL import Image
import torch
from transformers import AutoProcessor, AutoModelForCausalLM
```

- **requests**: Used to make HTTP requests. In this case, it downloads an image from a URL.
- **PIL (Pillow)**: Provides tools for manipulating images. Here, it's used to open the downloaded image.
- **torch**: PyTorch is imported to handle tensor operations and determine the hardware availability (CPU or GPU).
- **transformers**: This module provides easy access to Florence-2 by using `AutoProcessor` and `AutoModelForCausalLM` to load pre-trained models and process inputs.

Determining the Device and Data Type

```
device = (
    "cuda:0"
    if torch.cuda.is_available()
    else "cpu"
)

torch_dtype = (
    torch.float16
    if torch.cuda.is_available()
    else torch.float32
)
```

- **Device Setup**: The code checks if a CUDA-enabled GPU is available (`torch.cuda.is_available()`). The device is set to "cuda:0" if a GPU is available. Otherwise, it defaults to `"cpu"` (our case here).
- **Data Type Setup**: If a GPU is available, `torch.float16` is chosen, which uses half-precision floats to speed up processing and reduce memory usage. On the CPU, it defaults to `torch.float32` to maintain compatibility.

Loading the Model and Processor

```
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Florence-2-base",
    torch_dtype=torch_dtype,
    trust_remote_code=True
).to(device)

processor = AutoProcessor.from_pretrained(
    "microsoft/Florence-2-base",
    trust_remote_code=True
)
```

- **Model Initialization**:
  - `AutoModelForCausalLM.from_pretrained()` loads the pre-trained Florence-2 model from Microsoft's repository on Hugging Face. The `torch_dtype` is set according to the available hardware (GPU/CPU), and `trust_remote_code=True` allows the use of any custom code that might be provided with the model.
  - `.to(device)` moves the model to the appropriate device (either CPU or GPU). In our case, it will be set to `CPU`.
- **Processor Initialization**:
  - `AutoProcessor.from_pretrained()` loads the processor for Florence-2. The processor is responsible for transforming text and image inputs into a format the model can work with (e.g., encoding text, normalizing images, etc.).

Defining the Prompt

```
prompt = "<OD>"
```

- **Prompt Definition**: The string `"<OD>"` is used as a prompt. This refers to "Object Detection", instructing the model to detect objects on the image.

Downloading and Loading the Image

```
url = "https://huggingface.co/datasets/huggingface/"
      "documentation-images/resolve/main/transformers/"
      "tasks/car.jpg?download=true"
image = Image.open(requests.get(url, stream=True).raw)
```

- **Downloading the Image**: The `requests.get()` function fetches the image from the specified URL. The `stream=True` parameter ensures the image is streamed rather than downloaded completely at once.
- **Opening the Image**: `Image.open()` opens the image so the model can process it.

Processing Inputs

```
inputs = processor(
    text=prompt,
    images=image,
    return_tensors="pt"
).to(
    device,
    torch_dtype
)
```

- **Processing Input Data**: The `processor()` function processes the text (`prompt`) and the image (`image`). The `return_tensors="pt"` argument converts the processed data into PyTorch tensors, which are necessary for inputting data into the model.
- **Moving Inputs to Device**: `.to(device, torch_dtype)` moves the inputs to the correct device (CPU or GPU) and assigns the appropriate data type.

Generating the Output

```
generated_ids = model.generate(
    input_ids=inputs["input_ids"],
    pixel_values=inputs["pixel_values"],
    max_new_tokens=1024,
    do_sample=False,
    num_beams=3,
)
```

- **Model Generation**: `model.generate()` is used to generate the output based on the input data.
  - `input_ids`: Represents the tokenized form of the prompt.

  - `pixel_values`: Contains the processed image data.
  - `max_new_tokens=1024`: Specifies the maximum number of new tokens to be generated in the response. This limits the response length.
  - `do_sample=False`: Disables sampling; instead, the generation uses deterministic methods (beam search).
  - `num_beams=3`: Enables beam search with three beams, which improves output quality by considering multiple possibilities during generation.

Decoding the Generated Text

```
generated_text = processor.batch_decode(
    generated_ids,
    skip_special_tokens=False
)[0]
```

- **Batch Decode**: `processor.batch_decode()` decodes the generated IDs (tokens) into readable text. The `skip_special_tokens=False` parameter means that the output will include any special tokens that may be part of the response.

Post-processing the Generation

```
parsed_answer = processor.post_process_generation(
    generated_text,
    task="<OD>",
    image_size=(image.width, image.height)
)
```

- **Post-Processing**: `processor.post_process_generation()` is called to process the generated text further, interpreting it based on the task (`"<OD>"` for object detection) and the size of the image.
- This function extracts specific information from the generated text, such as bounding boxes for detected objects, making the output more useful for visual tasks.

Printing the Output

```
print(parsed_answer)
```

- Finally, `print(parsed_answer)` displays the output, which could include object detection results, such as bounding box coordinates and labels for the detected objects in the image.

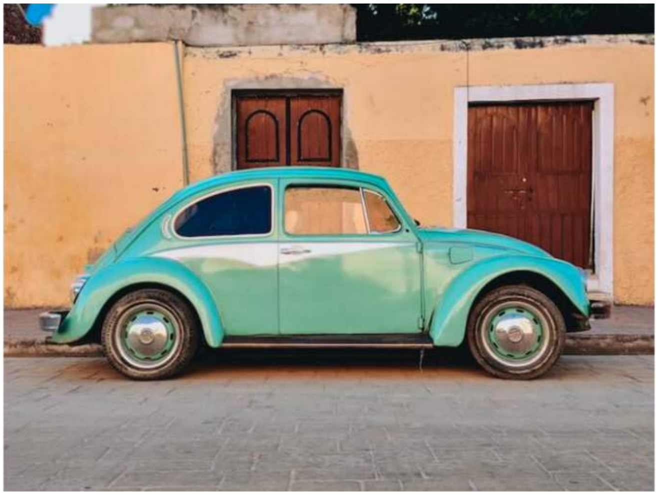
Result

Running the code, we get as the Parsed Answer:

```
[{'<OD>': {
   'bboxes': [
     [34.23999786376953, 160.0800018310547, 597.4400024414062],
     [371.7599792480469, 272.32000732421875, 241.67999267578125],
     [303.67999267578125, 247.4399871826172, 454.0799865722656],
     [276.7200012207031, 553.9199829101562, 370.79998779296875],
     [96.31999969482422, 280.55999755859375, 198.0800018310547],
     [371.2799987792969]
    ],
    'labels': ['car', 'door handle', 'wheel', 'wheel']
}}]
```

First, let's inspect the image:

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 8))
plt.imshow(image)
plt.axis('off')
plt.show()
```



By the Object Detection result, we can see that:

```
'labels': ['car', 'door handle', 'wheel', 'wheel']
```

It seems that at least a few objects were detected. We can also implement a code to draw the bounding boxes in the find objects:

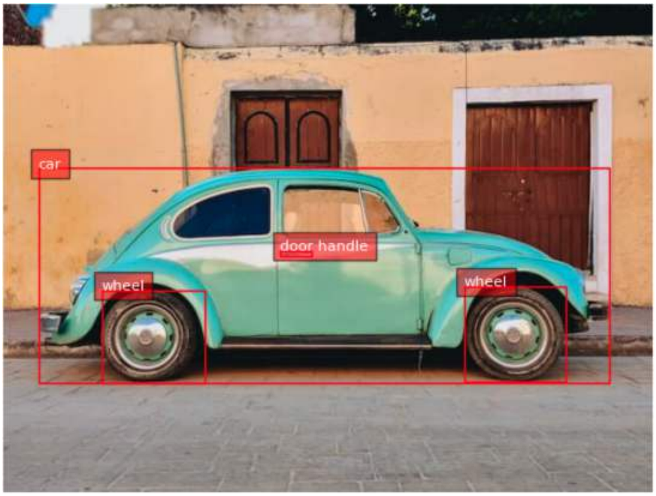
```
def plot_bbox(image, data):
   # Create a figure and axes
    fig, ax = plt.subplots()

    # Display the image
    ax.imshow(image)

    # Plot each bounding box
    for bbox, label in zip(data['bboxes'], data['labels']):
        # Unpack the bounding box coordinates
        x1, y1, x2, y2 = bbox
        # Create a Rectangle patch
        rect = patches.Rectangle(
           (x1, y1), x2 - x1, y2 - y1,
           linewidth=1,
           edgecolor='r',
           facecolor='none'
         )
        # Add the rectangle to the Axes
        ax.add_patch(rect)
        # Annotate the label
        plt.text(x1, y1, label, color='white', fontsize=8,
                 bbox=dict(facecolor='red', alpha=0.5))

    # Remove the axis ticks and labels
    ax.axis('off')

    # Show the plot
    plt.show()
```

> **Box (x0, y0, x1, y1)**: Location tokens correspond to the top-left and bottom-right corners of a box.

And running

```
plot_bbox(image, parsed_answer['<OD>'])
```

We get:



Florence-2 Tasks

Florence-2 is designed to perform a variety of computer vision and vision-language tasks through `prompts`. These tasks can be activated by providing a specific textual prompt to the model, as we saw with `<OD>` (Object Detection).

Florence-2's versatility comes from combining these prompts, allowing us to guide the model's behavior to perform specific vision tasks. Changing the prompt allows us to adapt Florence-2 to different tasks without needing task-specific modifications in the architecture. This capability directly results from Florence-2's unified model architecture and large-scale multi-task training on the FLD-5B dataset.

Here are some of the key tasks that Florence-2 can perform, along with example prompts:

Object Detection (OD)

- **Prompt**: `"<OD>"`
- **Description**: Identifies objects in an image and provides bounding boxes for each detected object. This task is helpful for applications like visual inspection, surveillance, and general object recognition.

Image Captioning

- **Prompt**: `"<CAPTION>"`
- **Description**: Generates a textual description for an input image. This task helps the model describe what is happening in the image, providing a human-readable caption for content understanding.

Detailed Captioning

- **Prompt**: `"<DETAILED_CAPTION>"`
- **Description**: Generates a more detailed caption with more nuanced information about the scene, such as the objects present and their relationships.

Visual Grounding

- **Prompt**: `"<CAPTION_TO_PHRASE_GROUNDING>"`
- **Description**: Links a textual description to specific regions in an image. For example, given a prompt like "a green car," the model highlights where the green car is in the image. This is useful for human-computer interaction, where you must find specific objects based on text.

Segmentation

- **Prompt**: `"<REFERRING_EXPRESSION_SEGMENTATION>"`
- **Description**: Performs segmentation based on a referring expression, such as "the blue cup." The model identifies and segments the specific region containing the object mentioned in the prompt (all related pixels).

Dense Region Captioning

- **Prompt**: `"<DENSE_REGION_CAPTION>"`
- **Description**: Provides captions for multiple regions within an image, offering a detailed breakdown of all visible areas, including different objects and their relationships.

OCR with Region

- **Prompt**: `"<OCR_WITH_REGION>"`
- **Description**: Performs Optical Character Recognition (OCR) on an image and provides bounding boxes for the detected text. This is useful for extracting and locating textual information in images, such as reading signs, labels, or other forms of text in images.

Phrase Grounding for Specific Expressions

- **Prompt**: `"<CAPTION_TO_PHRASE_GROUNDING>"` along with a specific expression, such as `"a wine glass"`.
- **Description**: Locates the area in the image that corresponds to a specific textual phrase. This task allows for identifying particular objects or elements when prompted with a word or keyword.

Open Vocabulary Object Detection

- **Prompt**: `"<OPEN_VOCABULARY_OD>"`
- **Description**: The model can detect objects without being restricted to a predefined list of classes, making it helpful in recognizing a broader range of items based on general visual understanding.

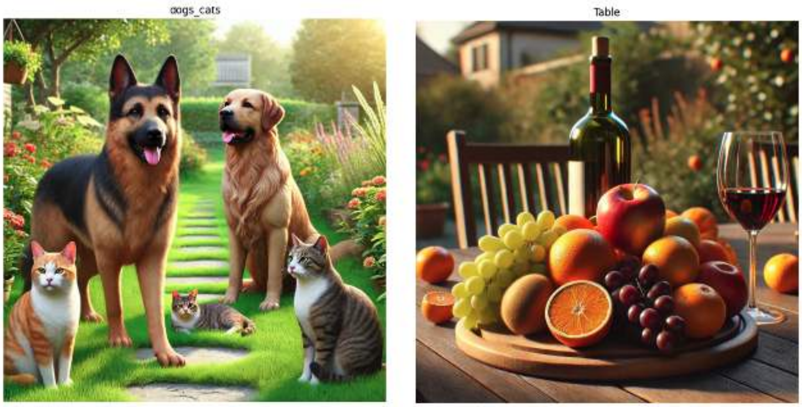
Exploring computer vision and vision-language tasks

For exploration, all codes can be found on the GitHub:

- 20-florence_2.ipynb

Let's use a couple of images created by Dall-E and upload them to the Rasp-5 (FileZilla can be used for that). The images will be saved on a sub-folder named `images` :

```
dogs_cats = Image.open('./images/dogs-cats.jpg')
table = Image.open('./images/table.jpg')
```



Let's create a function to facilitate our exploration and to keep track of the latency of the model for different tasks:

```
def run_example(task_prompt, text_input=None, image=None):
    start_time = time.perf_counter()  # Start timing
    if text_input is None:
        prompt = task_prompt
    else:
        prompt = task_prompt + text_input
    inputs = processor(text=prompt, images=image,
                       return_tensors="pt").to(device)
    generated_ids = model.generate(
      input_ids=inputs["input_ids"],
      pixel_values=inputs["pixel_values"],
      max_new_tokens=1024,
      early_stopping=False,
      do_sample=False,
      num_beams=3,
    )
    generated_text = processor.batch_decode(
       generated_ids,
       skip_special_tokens=False
    )[0]
    parsed_answer = processor.post_process_generation(
        generated_text,
        task=task_prompt,
        image_size=(image.width, image.height)
    )

    end_time = time.perf_counter() # End timing
    elapsed_time = end_time - start_time # Calculate elapsed time
    print(f" \n[INFO] ==> Florence-2-base ({task_prompt}), \
          took {elapsed_time:.1f} seconds to execute.\n")

    return parsed_answer
```

Caption

1. Dogs and Cats

```
run_example(task_prompt='<CAPTION>',image=dogs_cats)
```

```
[INFO] ==> Florence-2-base (<CAPTION>), \
took 16.1 seconds to execute.

{'<CAPTION>': 'A group of dogs and cats sitting in a garden.'}
```

2. Table

```
run_example(task_prompt='<CAPTION>',image=table)
```

```
[INFO] ==> Florence-2-base (<CAPTION>), \
took 16.5 seconds to execute.

{'<CAPTION>': 'A wooden table topped with a plate of fruit \
and a glass of wine.'}
```

DETAILED_CAPTION

1. Dogs and Cats

```
run_example(task_prompt='<DETAILED_CAPTION>',image=dogs_cats)
```

```
[INFO] ==> Florence-2-base (<DETAILED_CAPTION>), \
took 25.5 seconds to execute.

{'<DETAILED_CAPTION>': 'The image shows a group of cats and \
dogs sitting on top of a lush green field, surrounded by plants \
with flowers, trees, and a house in the background. The sky is \
visible above them, creating a peaceful atmosphere.'}
```

2. Table

```
run_example(task_prompt='<DETAILED_CAPTION>',image=table)
```

```
[INFO] ==> Florence-2-base (<DETAILED_CAPTION>), \
took 26.8 seconds to execute.

{'<DETAILED_CAPTION>': 'The image shows a wooden table with \
a bottle of wine and a glass of wine on it, surrounded by \
a variety of fruits such as apples, oranges, and grapes. \
In the background, there are chairs, plants, trees, and \
a house, all slightly blurred.'}
```

MORE_DETAILED_CAPTION

1. Dogs and Cats

```
run_example(task_prompt='<MORE_DETAILED_CAPTION>',image=dogs_cats)
```

```
[INFO] ==> Florence-2-base (<MORE_DETAILED_CAPTION>), \
took 49.8 seconds to execute.

{'<MORE_DETAILED_CAPTION>': 'The image shows a group of four \
cats and a dog in a garden. The garden is filled with colorful \
flowers and plants, and there is a pathway leading up to \
a house in the background. The main focus of the image is \
a large German Shepherd dog standing on the left side of \
the garden, with its tongue hanging out and its mouth open, \
as if it is panting or panting. On the right side, there are \
two smaller cats, one orange and one gray, sitting on the \
grass. In the background, there is another golden retriever \
dog sitting and looking at the camera. The sky is blue and \
the sun is shining, creating a warm and inviting atmosphere.'}
```

2. Table

```
run_example(task_prompt='< MORE_DETAILED_CAPTION>',image=table)
```

```
INFO] ==> Florence-2-base (<MORE_DETAILED_CAPTION>), \
took 32.4 seconds to execute.

{'<MORE_DETAILED_CAPTION>': 'The image shows a wooden table \
with a wooden tray on it. On the tray, there are various \
fruits such as grapes, oranges, apples, and grapes. There \
is also a bottle of red wine on the table. The background \
shows a garden with trees and a house. The overall mood \
of the image is peaceful and serene.'}
```

> We can note that the more detailed the caption task, the longer the latency and the possibility of mistakes (like "The image shows a group of four cats and a dog in a garden", instead of two dogs and three cats).

OD - Object Detection

We can run the same previous function for object detection using the prompt `<OD>`.

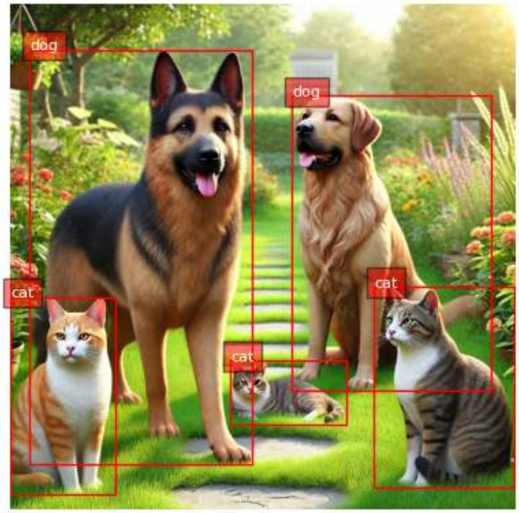
```
task_prompt = '<OD>'
results = run_example(task_prompt,image=dogs_cats)
print(results)
```

Let's see the result:

```
[INFO] ==> Florence-2-base (<OD>), took 20.9 seconds to execute.

{'<OD>': {'bboxes': [
  [737.79, 571.90, 1022.46, 980.48],
  [0.51, 593.40, 211.45, 991.74],
  [445.95, 721.40, 680.44, 850.43],
  [39.42, 91.64, 491.00, 933.37],
  [570.88, 184.83, 974.33, 782.84]
  ],
  'labels': ['cat', 'cat', 'cat', 'dog', 'dog']
}}
```

Only by the labels `['cat,' 'cat,' 'cat,' 'dog,' 'dog']` is it possible to see that the main objects in the image were captured. Let's apply the function used before to draw the bounding boxes:

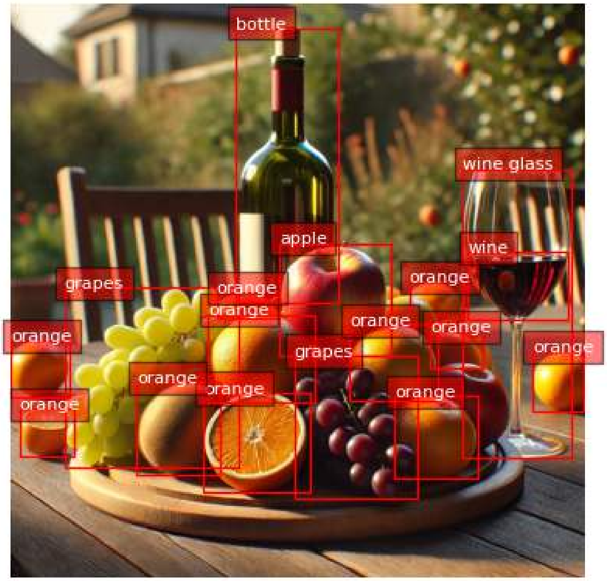
```
plot_bbox(dogs_cats, results['<OD>'])
```



Let's also do it with the Table image:

```
task_prompt = '<OD>'
results = run_example(task_prompt,image=table)
plot_bbox(table, results['<OD>'])
```

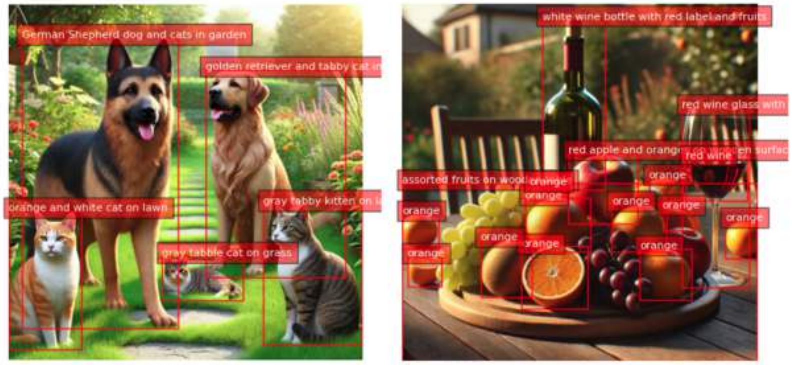
```
[INFO] ==> Florence-2-base (<OD>), took 40.8 seconds to execute.
```



DENSE_REGION_CAPTION

It is possible to mix the classic Object Detection with the Caption task in specific sub-regions of the image:

```
task_prompt = '<DENSE_REGION_CAPTION>'

results = run_example(task_prompt,image=dogs_cats)
plot_bbox(dogs_cats, results['<DENSE_REGION_CAPTION>'])

results = run_example(task_prompt,image=table)
plot_bbox(table, results['<DENSE_REGION_CAPTION>'])
```

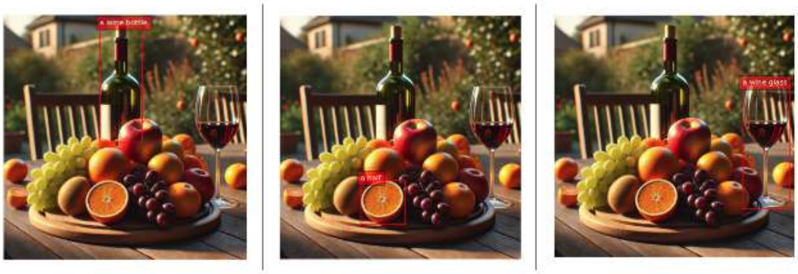


CAPTION_TO_PHRASE_GROUNDING

With this task, we can enter with a caption, such as “a wine glass”, “a wine bottle,” or “a half orange,” and Florence-2 will localize the object in the image:

```
task_prompt = '<CAPTION_TO_PHRASE_GROUNDING>'

results = run_example(
    task_prompt,
    text_input="a wine bottle",
    image=table
)
plot_bbox(table, results['<CAPTION_TO_PHRASE_GROUNDING>'])

results = run_example(
    task_prompt,
    text_input="a wine glass",
    image=table
)
plot_bbox(table, results['<CAPTION_TO_PHRASE_GROUNDING>'])
```

```
results = run_example(
    task_prompt,
    text_input="a half orange",
    image=table
)
plot_bbox(table, results['<CAPTION_TO_PHRASE_GROUNDING>'])
```



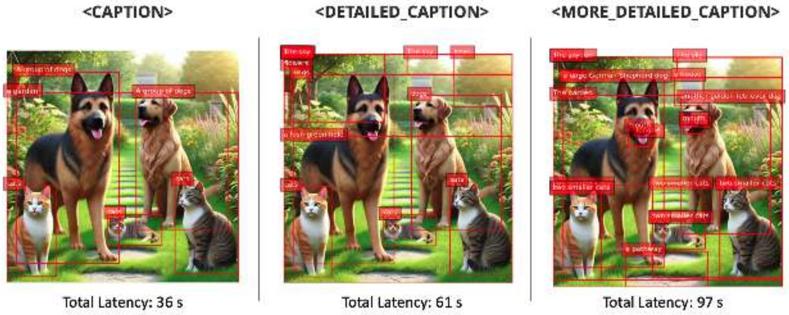
```
[INFO] ==> Florence-2-base (<CAPTION_TO_PHRASE_GROUNDING>), \
took 15.7 seconds to execute
each task.
```

Cascade Tasks

We can also enter the image caption as the input text to push Florence-2 to find more objects:

```
task_prompt = '<CAPTION>'
results = run_example(task_prompt,image=dogs_cats)
text_input = results[task_prompt]
task_prompt = '<CAPTION_TO_PHRASE_GROUNDING>'
results = run_example(task_prompt, text_input,image=dogs_cats)
plot_bbox(dogs_cats, results['<CAPTION_TO_PHRASE_GROUNDING>'])
```

Changing the task_prompt among <CAPTION,> <DETAILED_CAPTION> and <MORE_DETAILED_CAPTION>, we will get more objects in the image.

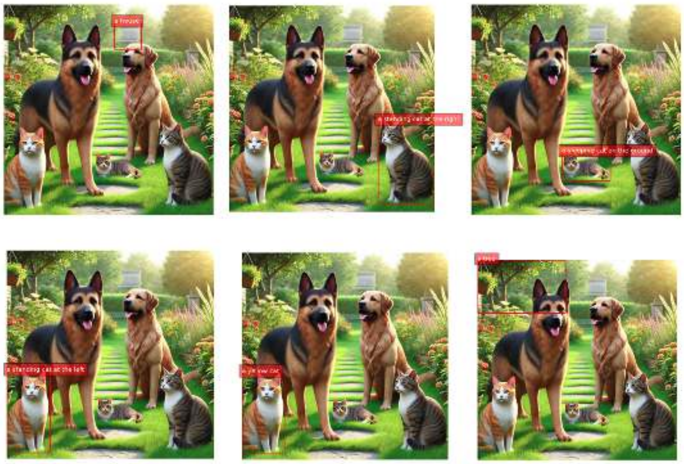


OPEN_VOCABULARY_DETECTION

<OPEN_VOCABULARY_DETECTION> allows Florence-2 to detect recognizable objects in an image without relying on a predefined list of categories, making it a versatile tool for identifying various items that may not have been explicitly labeled during training. Unlike <CAPTION_TO_PHRASE_GROUNDING>, which requires a specific text phrase to locate and highlight a particular object in an image, <OPEN_VOCABULARY_DETECTION> performs a broad scan to find and classify all objects present.

This makes <OPEN_VOCABULARY_DETECTION> particularly useful for applications where you need a comprehensive overview of everything in an image without prior knowledge of what to expect. Enter with a text describing specific objects not previously detected, resulting in their detection. For example:

```
task_prompt = '<OPEN_VOCABULARY_DETECTION>'

text = [
    "a house",
    "a tree",
    "a standing cat at the left",
    "a sleeping cat on the ground",
    "a standing cat at the right",
    "a yellow cat"
]

for txt in text:
    results = run_example(
        task_prompt,
        text_input=txt,
        image=dogs_cats
    )

    bbox_results = convert_to_od_format(
        results['<OPEN_VOCABULARY_DETECTION>']
    )

    plot_bbox(dogs_cats, bbox_results)
```



```
[INFO] ==> Florence-2-base (<OPEN_VOCABULARY_DETECTION>), \
took 15.1 seconds to execute
each task.
```

> Note: Trying to use Florence-2 to find objects that were not found can leads to mistakes (see exaamples on the Notebook).

Referring expression segmentation

We can also segment a specific object in the image and give its description (caption), such as “a wine bottle” on the table image or “a German Sheppard” on the dogs_cats.

Referring expression segmentation results format: `{'<REFERRING_EXPRESSION_SEGMENTATION>': {'Polygons': [[[polygon]], ...], 'labels': ['', '', ...]}}`, one object is represented by a list of polygons. each polygon is `[x1, y1, x2, y2, ..., xn, yn]`.

> **Polygon (x1, y1, …, xn, yn)**: Location tokens represent the vertices of a polygon in clockwise order.

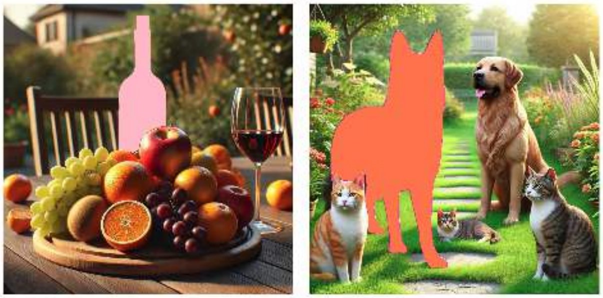
So, let’s first create a function to plot the segmentation:

```python
from PIL import Image, ImageDraw, ImageFont
import copy
import random
import numpy as np
colormap = [
    'blue', 'orange', 'green', 'purple', 'brown', 'pink', 'gray',
    'olive', 'cyan', 'red', 'lime', 'indigo', 'violet', 'aqua',
    'magenta', 'coral', 'gold', 'tan', 'skyblue'
]
```

```
def draw_polygons(image, prediction, fill_mask=False):
    """
    Draws segmentation masks with polygons on an image.

    Parameters:
      - image_path: Path to the image file.
      - prediction: Dictionary containing 'polygons' and 'labels'
              keys. 'polygons' is a list of lists, each
              containing vertices of a polygon. 'labels' is
              a list of labels corresponding to each polygon.
- fill_mask: Boolean indicating whether to fill the polygons
             with color.
    """
    # Load the image

    draw = ImageDraw.Draw(image)


    # Set up scale factor if needed (use 1 if not scaling)
    scale = 1

    # Iterate over polygons and labels
    for polygons, label in zip(
        prediction['polygons'],
        prediction['labels']
    ):
        color = random.choice(colormap)
        fill_color = (
            random.choice(colormap)
            if fill_mask else None
         )

        for _polygon in polygons:
            _polygon = np.array(_polygon).reshape(-1, 2)
            if len(_polygon) < 3:
                print('Invalid polygon:', _polygon)
                continue

            _polygon = (_polygon * scale).reshape(-1).tolist()

            # Draw the polygon
            if fill_mask:
               draw.polygon(
                   _polygon,
                  outline=color,
                  fill=fill_color
               )
            else:
               draw.polygon(
               _polygon,
               outline=color
               )

            # Draw the label text
            draw.text(
               (_polygon[0] + 8, _polygon[1] + 2),
               label,
               fill=color
            )

    # Save or display the image
    #image.show()  # Display the image
    display(image)
```

Now we can run the functions:

```
task_prompt = '<REFERRING_EXPRESSION_SEGMENTATION>'

results = run_example(
    task_prompt,
    text_input="a wine bottle",
    image=table
)
output_image = copy.deepcopy(table)
draw_polygons(output_image,
              results['<REFERRING_EXPRESSION_SEGMENTATION>'],
              fill_mask=True)

results = run_example(
    task_prompt,
    text_input="a german sheppard",
    image=dogs_cats
)
output_image = copy.deepcopy(dogs_cats)
draw_polygons(output_image,
              results['<REFERRING_EXPRESSION_SEGMENTATION>'],
              fill_mask=True)
```



```
[INFO] ==> Florence-2-base
(<REFERRING_EXPRESSION_SEGMENTATION>), took 207.0 seconds
to execute each task.
```

Region to Segmentation

With this task, it is also possible to give the object coordinates in the image to segment it. The input format is `'<loc_x1><loc_y1><loc_x2><loc_y2>'`, `[x1, y1, x2, y2]` , which is the quantized coordinates in [0, 999].

For example, when running the code:

```
task_prompt = '<CAPTION_TO_PHRASE_GROUNDING>'
results = run_example(
    task_prompt,
    text_input="a half orange",
    image=table
)
results
```

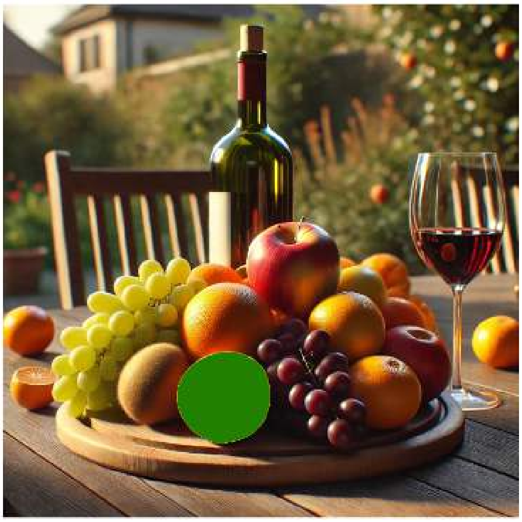
The results were:

```
{'<CAPTION_TO_PHRASE_GROUNDING>': {'bboxes': [[343.552001953125,
    689.6640625,
    530.9440307617188,
    873.9840698242188]],
  'labels': ['a half']}}
```

Using the bboxes rounded coordinates:

```
task_prompt = '<REGION_TO_SEGMENTATION>'
results = run_example(
    task_prompt,
    text_input=(
        "<loc_343><loc_690>"
        "<loc_531><loc_874>"
    ),
    image=table
)
output_image = copy.deepcopy(table)
draw_polygons(
    output_image,
    results['<REGION_TO_SEGMENTATION>'],
    fill_mask=True
)
```

We got the segmentation of the object on those coordinates (Latency: 83 seconds):



Region to Texts

We can also give the region (coordinates and ask for a caption):

```
task_prompt = '<REGION_TO_CATEGORY>'
results = run_example(
    task_prompt,
    text_input=(
        "<loc_343><loc_690>"
        "<loc_531><loc_874>"
    ),
    image=table
)
results
```

```
[INFO] ==> Florence-2-base (<REGION_TO_CATEGORY>), \
took 14.3 seconds to execute.

{{
  '<REGION_TO_CATEGORY>':
    'orange<loc_343><loc_690>'
    '<loc_531><loc_874>'
}
```

The model identified an orange in that region. Let's ask for a description:

```
task_prompt = '<REGION_TO_DESCRIPTION>'
results = run_example(
    task_prompt,
    text_input=(
        "<loc_343><loc_690>"
        "<loc_531><loc_874>"
    ),
    image=table
)
results
```

```
[INFO] ==> Florence-2-base (<REGION_TO_CATEGORY>), \
took 14.6 seconds to execute.

{
  '<REGION_TO_CATEGORY>':
    'orange<loc_343><loc_690>'
    '<loc_531><loc_874>'
}
```

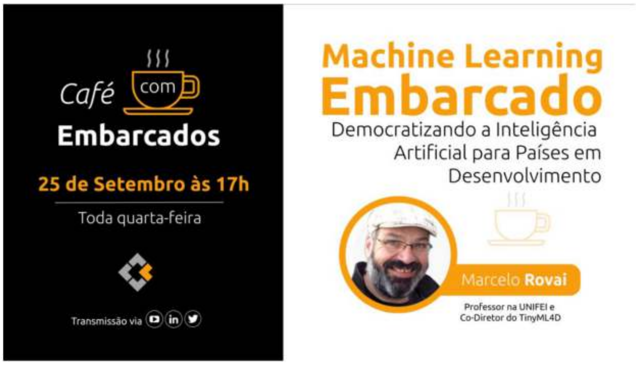
In this case, the description did not provide more details, but it could. Try another example.

OCR

With Florence-2, we can perform Optical Character Recognition (OCR) on an image, getting what is written on it (`task_prompt = '<OCR>'` and also get the bounding boxes (location) for the detected text (`ask_prompt = '<OCR_WITH_REGION>'`). Those tasks can help extract and locate textual information in images, such as reading signs, labels, or other forms of text in images.

Let's upload a flyer from a talk in Brazil to Raspi. Let's test works in another language, here Portuguese):

```
flayer = Image.open('./images/embarcados.jpg')
# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(flayer)
plt.axis('off')
#plt.title("Image")
plt.show()
```



Let's examine the image with `'<MORE_DETAILED_CAPTION>'`:

```
[INFO] ==> Florence-2-base (<MORE_DETAILED_CAPTION>), \
took 85.2 seconds to execute.

{'<MORE_DETAILED_CAPTION>': 'The image is a promotional poster \
for an event called "Machine Learning Embarcados" hosted by \
Marcelo Roval. The poster has a black background with white \
text. On the left side of the poster, there is a logo of a \
coffee cup with the text "Café Com Embarcados" above it. \
Below the logo, it says "25 de Setembro as 17th" which \
translates to "25th of September as 17" in English. \n\nOn \
the right side, there aretwo smaller text boxes with the names \
of the participants and their names. The first text box reads \
"Democratizando a Inteligência Artificial para Paises em \
Desenvolvimento" and the second text box says "Toda \
quarta-feira" which is Portuguese for "Transmissão via in \
Portuguese".\n\nIn the center of the image, there has a photo \
of Marcelo, a man with a beard and glasses, smiling at the \
camera. He is wearing a white hard hat and a white shirt. \
The text boxes are in orange and yellow colors.'}
```

The description is very accurate. Let's get to the more important words with the task OCR:

```
task_prompt = '<OCR>'
run_example(task_prompt,image=flayer)
```

```
[INFO] ==> Florence-2-base (<OCR>), took 37.7 seconds to execute.

{'<OCR>':
 'Machine Learning Café com Embarcado Embarcados '
 'Democratizando a Inteligência Artificial para Paises em '
 '25 de Setembro às 17h Desenvolvimento Toda quarta-feira '
 'Marcelo Roval Professor na UNIFIEI e Transmissão via in '
 'Co-Director do TinyML4D'}
```

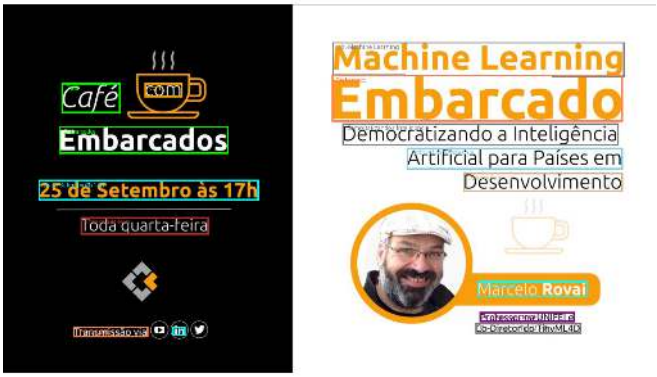
Let's locate the words in the flyer:

```
task_prompt = '<OCR_WITH_REGION>'
results = run_example(task_prompt,image=flayer)
```

Let's also create a function to draw bounding boxes around the detected words:

```
def draw_ocr_bboxes(image, prediction):
    scale = 1
    draw = ImageDraw.Draw(image)
    bboxes = prediction['quad_boxes']
    labels = prediction['labels']
    for box, label in zip(bboxes, labels):
        color = random.choice(colormap)
        new_box = (np.array(box) * scale).tolist()
        draw.polygon(new_box, width=3, outline=color)
        draw.text((new_box[0]+8, new_box[1]+2),
                    "{}".format(label),
                    align="right",
                    fill=color)
    display(image)
```

```
output_image = copy.deepcopy(flayer)
draw_ocr_bboxes(output_image, results['<OCR_WITH_REGION>'])
```



We can inspect the detected words:

```
results['<OCR_WITH_REGION>']['labels']
```

```
'</s>Machine Learning',
 'Café',
 'com',
 'Embarcado',
 'Embarcados',
 'Democratizando a Inteligência',
 'Artificial para Paises em',
 '25 de Setembro ás 17h',
 'Desenvolvimento',
 'Toda quarta-feira',
 'Marcelo Roval',
 'Professor na UNIFIEI e',
 'Transmissão via',
 'in',
 'Co-Director do TinyML4D']
```

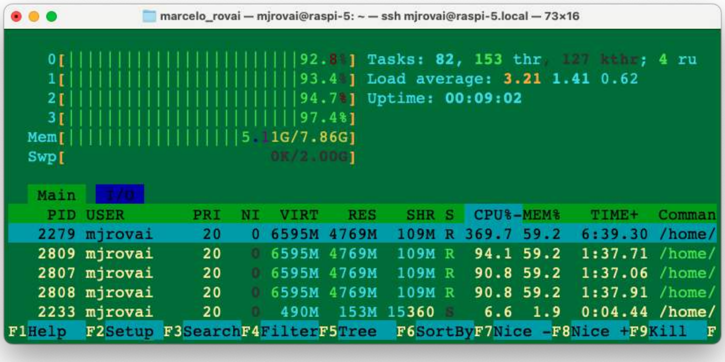
Latency Summary

The latency observed for different tasks using Florence-2 on the Raspberry Pi (Raspi-5) varied depending on the complexity of the task:

- **Image Captioning**: It took approximately 16-17 seconds to generate a caption for an image.
- **Detailed Captioning**: Increased latency to around 25-27 seconds, requiring generating more nuanced scene descriptions.
- **More Detailed Captioning**: It took about 32-50 seconds, and the latency increased as the description grew more complex.
- **Object Detection**: It took approximately 20-41 seconds, depending on the image's complexity and the number of detected objects.

- **Visual Grounding**: Approximately 15-16 seconds to localize specific objects based on textual prompts.
- **OCR (Optical Character Recognition)**: Extracting text from an image took around 37-38 seconds.
- **Segmentation and Region to Segmentation**: Segmentation tasks took considerably longer, with a latency of around 83-207 seconds, depending on the complexity and the number of regions to be segmented.

These latency times highlight the resource constraints of edge devices like the Raspberry Pi and emphasize the need to optimize the model and the environment to achieve real-time performance.



marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@raspi-5.local — 73×16

```
    0[||||||||||||||||||||||||92.8%] Tasks: 82, 153 thr, 127 kthr; 4 ru
    1[||||||||||||||||||||||||93.4%] Load average: 3.21 1.41 0.62
    2[||||||||||||||||||||||||94.7%] Uptime: 00:09:02
    3[||||||||||||||||||||||||97.4%]
  Mem[||||||||||||||||||5.11G/7.86G]
  Swp[                     0K/2.00G]

   Main  I/O
    PID USER       PRI  NI  VIRT   RES   SHR S  CPU%-MEM%   TIME+  Comman
   2279 mjrovai     20   0 6595M 4769M  109M R 369.7 59.2  6:39.30 /home/
   2809 mjrovai     20   0 6595M 4769M  109M R  94.1 59.2  1:37.71 /home/
   2807 mjrovai     20   0 6595M 4769M  109M R  90.8 59.2  1:37.06 /home/
   2808 mjrovai     20   0 6595M 4769M  109M R  90.8 59.2  1:37.91 /home/
   2233 mjrovai     20   0  490M  153M 15360 S   6.6  1.9  0:04.44 /home/
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F
```

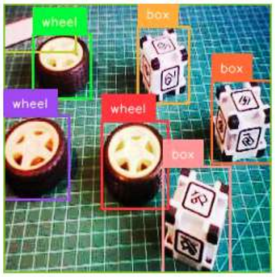
> Running complex tasks can use all 8 GB of the Raspi-5's memory. For example, the above screenshot during the Florence OD task shows 4 CPUs at full speed and over 5 GB of memory in use. Consider increasing the SWAP memory to 2 GB.

Checking the CPU temperature with `vcgencmd measure_temp` , showed that temperature can go up to +80oC.

Fine-Tunning

As explored in this lab, Florence supports many tasks out of the box, including captioning, object detection, OCR, and more. However, like other pre-trained foundational models, Florence-2 may need domain-specific knowledge. For example, it may need to improve with medical or satellite imagery. In such cases, **fine-tuning** with a custom dataset is necessary. The Roboflow tutorial, How to Fine-tune Florence-2 for Object Detection Tasks, shows how to fine-tune Florence-2 on object detection datasets to improve model performance for our specific use case.

Based on the above tutorial, it is possible to fine-tune the Florence-2 model to detect boxes and wheels used in previous labs:



```
{"<OD>": {"bboxes": [[0.1599999964237213, 133.59999084472656,
78.23999786376953, 232.1599884033203], [117.27999877929688,
139.0399932861328, 196.63999938964844, 243.67999267578125],
[190.239990234375, 193.1199951171875, 270.239990234375, 319.5199890136719],
[248.1599884033203, 91.04000091552734, 319.5199890136719,
189.27999877929688], [160.8000030517578, 27.68000030517578,
221.27999877929688, 118.23999786376953], [0.1599999964237213,
0.1599999964237213, 86.23999786376953, 57.119998931884766],
[35.36000061035156, 36.31999969482422, 104.15999603271484,
112.15999603271484], [0.1599999964237213, 0.47999998927116394,
319.5199890136719, 319.5199890136719]], "labels": ["wheel", "wheel", "box", "box",
"box", "box", "wheel", "box"]}}
```

It is important to note that after fine-tuning, the model can still detect classes that don't belong to our custom dataset, like cats, dogs, grapes, etc, as seen before).

The complete fine-tunning project using a previously annotated dataset in Roboflow and executed on CoLab can be found in the notebook:

- 30-Finetune_florence_2_on_detection_dataset_box_vs_wheel.ipynb

In another example, in the post, Fine-tuning Florence-2 - Microsoft's Cutting-edge Vision Language Models, the authors show an example of fine-tuning Florence on `DocVQA`. The authors report that Florence 2 can perform visual question answering (VQA), but the released models don't include VQA capability.

Conclusion

Florence-2 offers a versatile and powerful approach to vision-language tasks at the edge, providing performance that rivals larger, task-specific models, such as YOLO for object detection, BERT/RoBERTa for text analysis, and specialized OCR models.

Thanks to its multi-modal transformer architecture, Florence-2 is more flexible than YOLO in terms of the tasks it can handle. These include object detection, image captioning, and visual grounding.

Unlike **BERT**, which focuses purely on language, Florence-2 integrates vision and language, allowing it to excel in applications that require both modalities, such as image captioning and visual grounding.

Moreover, while traditional **OCR models** such as Tesseract and EasyOCR are designed solely for recognizing and extracting text from images, Florence-2's OCR capabilities are part of a broader framework that includes contextual understanding and visual-text alignment. This makes it particularly useful for scenarios that require both reading text and interpreting its context within images.

Overall, Florence-2 stands out for its ability to seamlessly integrate various vision-language tasks into a unified model that is efficient enough to run on edge devices like the Raspberry Pi. This makes it a compelling choice for developers and researchers exploring AI applications at the edge.

Key Advantages of Florence-2

1. **Unified Architecture**

- Single model handles multiple vision tasks vs. specialized models (YOLO, BERT, Tesseract)
- Eliminates the need for multiple model deployments and integrations
- Consistent API and interface across tasks

2. **Performance Comparison**
   - Object Detection: Comparable to YOLOv8 (~37.5 mAP on COCO vs. YOLOv8's ~39.7 mAP) despite being general-purpose
   - Text Recognition: Handles multiple languages effectively like specialized OCR models (Tesseract, EasyOCR)
   - Language Understanding: Integrates BERT-like capabilities for text processing while adding visual context
3. **Resource Efficiency**
   - The Base model (232M parameters) achieves strong results despite smaller size
   - Runs effectively on edge devices (Raspberry Pi)
   - Single model deployment vs. multiple specialized models

Trade-offs

1. **Performance vs. Specialized Models**
   - YOLO series may offer faster inference for pure object detection
   - Specialized OCR models might handle complex document layouts better
   - BERT/RoBERTa provide deeper language understanding for text-only tasks
2. **Resource Requirements**
   - Higher latency on edge devices (15-200s depending on task)
   - Requires careful memory management on Raspberry Pi
   - It may need optimization for real-time applications
3. **Deployment Considerations**
   - Initial setup is more complex than single-purpose models
   - Requires understanding of multiple task types and prompts
   - The learning curve for optimal prompt engineering

Best Use Cases

1. **Resource-Constrained Environments**
   - Edge devices requiring multiple vision capabilities
   - Systems with limited storage/deployment capacity
   - Applications needing flexible vision processing

2. **Multi-modal Applications**
    - Content moderation systems
    - Accessibility tools
    - Document analysis workflows
3. **Rapid Prototyping**
    - Quick deployment of vision capabilities
    - Testing multiple vision tasks without separate models
    - Proof-of-concept development

Future Implications

Florence-2 represents a shift toward unified vision models that could eventually replace task-specific architectures in many applications. While specialized models maintain advantages in specific scenarios, the convenience and efficiency of unified models like Florence-2 make them increasingly attractive for real-world deployments.

The lab demonstrates Florence-2's viability on edge devices, suggesting future IoT, mobile computing, and embedded systems applications where deploying multiple specialized models would be impractical.

Resources

- 10-florence2_test.ipynb
- 20-florence_2.ipynb
- 30-Finetune_florence_2_on_detection_dataset_box_vs_wheel.ipynb

Shared Labs

The labs in this section cover topics and techniques that are applicable across different hardware platforms. These labs are designed to be independent of specific boards, allowing you to focus on the fundamental concepts and algorithms used in (tiny) ML applications.

By exploring these shared labs, you'll gain a deeper understanding of the common challenges and solutions in embedded machine learning. The knowledge and skills acquired here will be valuable regardless of the specific hardware you work with in the future.

Exercise	Nicla Vision	XIAO ESP32S3
KWS Feature Engineering	Link	Link
DSP Spectral Features Block	Link	Link

KWS Feature Engineering

Figure 20.26: *DALL·E 3 Prompt: 1950s style cartoon scene set in an audio research room. Two scientists, one holding a magnifying glass and the other taking notes, examine large charts pinned to the wall. These charts depict FFT graphs and time curves related to audio data analysis. The room has a retro ambiance, with wooden tables, vintage lamps, and classic audio analysis tools.*

Overview

In this hands-on tutorial, the emphasis is on the critical role that feature engineering plays in optimizing the performance of machine learning models applied to audio classification tasks, such as speech recognition. It is essential to be aware that the performance of any machine learning model relies heavily on the quality of features used, and we will deal with "under-the-hood" mechanics of feature extraction, mainly focusing on Mel-frequency Cepstral Coefficients (MFCCs), a cornerstone in the field of audio signal processing.

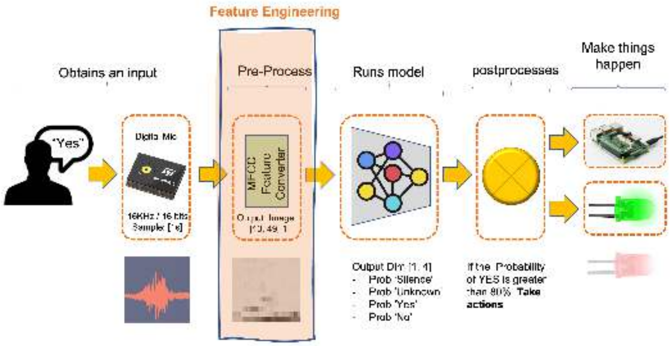
Machine learning models, especially traditional algorithms, don't understand audio waves. They understand numbers arranged in some meaningful way, i.e., features. These features encapsulate the characteristics of the audio signal, making it easier for models to distinguish between different sounds.

> This tutorial will deal with generating features specifically for audio classification. This can be particularly interesting for applying machine learning to a variety of audio data, whether for speech recognition, music categorization, insect classification based on wingbeat sounds, or other sound analysis tasks

The KWS

The most common TinyML application is Keyword Spotting (KWS), a subset of the broader field of speech recognition. While general speech recognition transcribes all spoken words into text, Keyword Spotting focuses on detecting specific "keywords" or "wake words" in a continuous audio stream. The system is trained to recognize these keywords as predefined phrases or words, such as *yes* or *no*. In short, KWS is a specialized form of speech recognition with its own set of challenges and requirements.

Here a typical KWS Process using MFCC Feature Converter:



Applications of KWS

- **Voice Assistants**: In devices like Amazon's Alexa or Google Home, KWS is used to detect the wake word ("Alexa" or "Hey Google") to activate the device.
- **Voice-Activated Controls**: In automotive or industrial settings, KWS can be used to initiate specific commands like "Start engine" or "Turn off lights."
- **Security Systems**: Voice-activated security systems may use KWS to authenticate users based on a spoken passphrase.

- **Telecommunication Services**: Customer service lines may use KWS to route calls based on spoken keywords.

Differences from General Speech Recognition

- **Computational Efficiency**: KWS is usually designed to be less computationally intensive than full speech recognition, as it only needs to recognize a small set of phrases.
- **Real-time Processing**: KWS often operates in real-time and is optimized for low-latency detection of keywords.
- **Resource Constraints**: KWS models are often designed to be lightweight, so they can run on devices with limited computational resources, like microcontrollers or mobile phones.
- **Focused Task**: While general speech recognition models are trained to handle a broad range of vocabulary and accents, KWS models are fine-tuned to recognize specific keywords, often in noisy environments accurately.

Overview to Audio Signals

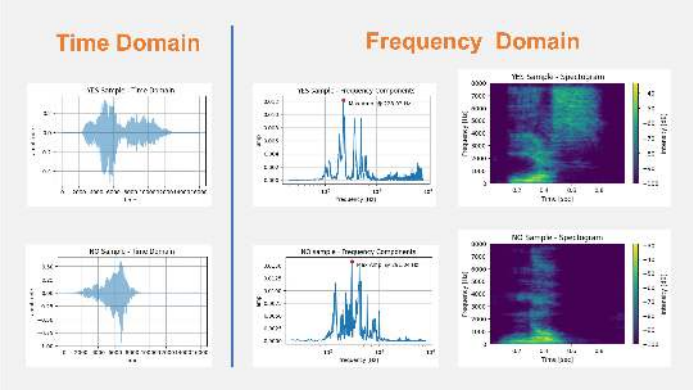
Understanding the basic properties of audio signals is crucial for effective feature extraction and, ultimately, for successfully applying machine learning algorithms in audio classification tasks. Audio signals are complex waveforms that capture fluctuations in air pressure over time. These signals can be characterized by several fundamental attributes: sampling rate, frequency, and amplitude.

- **Frequency and Amplitude**: Frequency refers to the number of oscillations a waveform undergoes per unit time and is also measured in Hz. In the context of audio signals, different frequencies correspond to different pitches. Amplitude, on the other hand, measures the magnitude of the oscillations and correlates with the loudness of the sound. Both frequency and amplitude are essential features that capture audio signals' tonal and rhythmic qualities.
- **Sampling Rate**: The sampling rate, often denoted in Hertz (Hz), defines the number of samples taken per second when digitizing an analog signal. A higher sampling rate allows for a more accurate digital representation of the signal but also demands more computational resources for processing. Typical sampling rates include 44.1 kHz for CD-quality audio and 16 kHz or 8 kHz for speech recognition tasks. Understanding the trade-offs in selecting an appropriate sampling rate is essential for balancing accuracy and computational efficiency. In general, with TinyML projects, we work with 16 kHz. Although music tones can be heard at frequencies up to 20 kHz, voice maxes out at 8 kHz. Traditional telephone systems use an 8 kHz sampling frequency.

  For an accurate representation of the signal, the sampling rate must be at least twice the highest frequency present in the signal.

- **Time Domain vs. Frequency Domain**: Audio signals can be analyzed in the time and frequency domains. In the time domain, a signal is represented as a waveform where the amplitude is plotted against time. This representation helps to observe temporal features like onset and duration but the signal's tonal characteristics are not well evidenced. Conversely, a frequency domain representation provides a view of the signal's constituent frequencies and their respective amplitudes, typically obtained via a Fourier Transform. This is invaluable for tasks that require understanding the signal's spectral content, such as identifying musical notes or speech phonemes (our case).

The image below shows the words `YES` and `NO` with typical representations in the Time (Raw Audio) and Frequency domains:



Why Not Raw Audio?

While using raw audio data directly for machine learning tasks may seem tempting, this approach presents several challenges that make it less suitable for building robust and efficient models.

Using raw audio data for Keyword Spotting (KWS), for example, on TinyML devices poses challenges due to its high dimensionality (using a 16 kHz sampling rate), computational complexity for capturing temporal features, susceptibility to noise, and lack of semantically meaningful features, making feature extraction techniques like MFCCs a more practical choice for resource-constrained applications.

Here are some additional details of the critical issues associated with using raw audio:

- **High Dimensionality**: Audio signals, especially those sampled at high rates, result in large amounts of data. For example, a 1-second audio clip sampled at 16 kHz will have 16,000 individual data points. High-dimensional data increases computational complexity, leading to longer training times and higher computational costs, making it impractical for resource-constrained environments. Furthermore, the wide dynamic range of audio signals requires a significant amount of bits per sample, while conveying little useful information.

- **Temporal Dependencies**: Raw audio signals have temporal structures that simple machine learning models may find hard to capture. While recurrent neural networks like LSTMs can model such dependencies, they are computationally intensive and tricky to train on tiny devices.
- **Noise and Variability**: Raw audio signals often contain background noise and other non-essential elements affecting model performance. Additionally, the same sound can have different characteristics based on various factors such as distance from the microphone, the orientation of the sound source, and acoustic properties of the environment, adding to the complexity of the data.
- **Lack of Semantic Meaning**: Raw audio doesn't inherently contain semantically meaningful features for classification tasks. Features like pitch, tempo, and spectral characteristics, which can be crucial for speech recognition, are not directly accessible from raw waveform data.
- **Signal Redundancy**: Audio signals often contain redundant information, with certain portions of the signal contributing little to no value to the task at hand. This redundancy can make learning inefficient and potentially lead to overfitting.

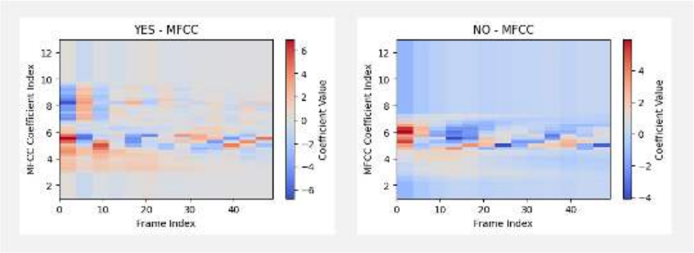
For these reasons, feature extraction techniques such as Mel-frequency Cepstral Coefficients (MFCCs), Mel-Frequency Energies (MFEs), and simple Spectograms are commonly used to transform raw audio data into a more manageable and informative format. These features capture the essential characteristics of the audio signal while reducing dimensionality and noise, facilitating more effective machine learning.

Overview to MFCCs

What are MFCCs?

Mel-frequency Cepstral Coefficients (MFCCs) are a set of features derived from the spectral content of an audio signal. They are based on human auditory perceptions and are commonly used to capture the phonetic characteristics of an audio signal. The MFCCs are computed through a multi-step process that includes pre-emphasis, framing, windowing, applying the Fast Fourier Transform (FFT) to convert the signal to the frequency domain, and finally, applying the Discrete Cosine Transform (DCT). The result is a compact representation of the original audio signal's spectral characteristics.

The image below shows the words `YES` and `NO` in their MFCC representation:



This video explains the Mel Frequency Cepstral Coefficients (MFCC) and how to compute them.

Why are MFCCs important?

MFCCs are crucial for several reasons, particularly in the context of Keyword Spotting (KWS) and TinyML:

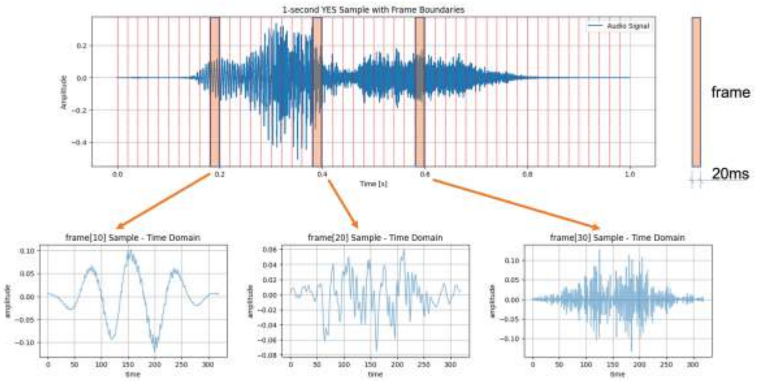
- **Dimensionality Reduction**: MFCCs capture essential spectral characteristics of the audio signal while significantly reducing the dimensionality of the data, making it ideal for resource-constrained TinyML applications.
- **Robustness**: MFCCs are less susceptible to noise and variations in pitch and amplitude, providing a more stable and robust feature set for audio classification tasks.
- **Human Auditory System Modeling**: The Mel scale in MFCCs approximates the human ear's response to different frequencies, making them practical for speech recognition where human-like perception is desired.
- **Computational Efficiency**: The process of calculating MFCCs is computationally efficient, making it well-suited for real-time applications on hardware with limited computational resources.

In summary, MFCCs offer a balance of information richness and computational efficiency, making them popular for audio classification tasks, particularly in constrained environments like TinyML.

Computing MFCCs

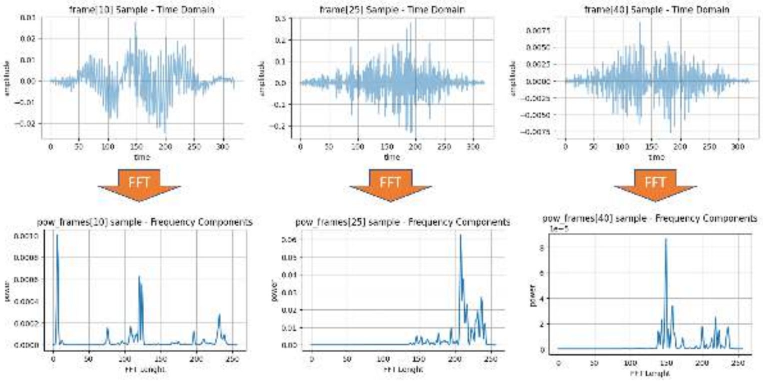
The computation of Mel-frequency Cepstral Coefficients (MFCCs) involves several key steps. Let's walk through these, which are particularly important for Keyword Spotting (KWS) tasks on TinyML devices.

- **Pre-emphasis**: The first step is pre-emphasis, which is applied to accentuate the high-frequency components of the audio signal and balance the frequency spectrum. This is achieved by applying a filter that amplifies the difference between consecutive samples. The formula for pre-emphasis is: $y(t) = x(t) - \alpha x(t-1)$, where α is the pre-emphasis factor, typically around 0.97.
- **Framing**: Audio signals are divided into short frames (the *frame length*), usually 20 to 40 milliseconds. This is based on the assumption that frequencies in a signal are stationary over a short period. Framing helps in analyzing the signal in such small time slots. The *frame stride* (or step) will displace one frame and the adjacent. Those steps could be sequential or overlapped.
- **Windowing**: Each frame is then windowed to minimize the discontinuities at the frame boundaries. A commonly used window function is the Hamming window. Windowing prepares the signal for a Fourier transform by minimizing the edge effects. The image below shows three frames (10, 20, and 30) and the time samples after windowing (note that the frame length and frame stride are 20 ms):

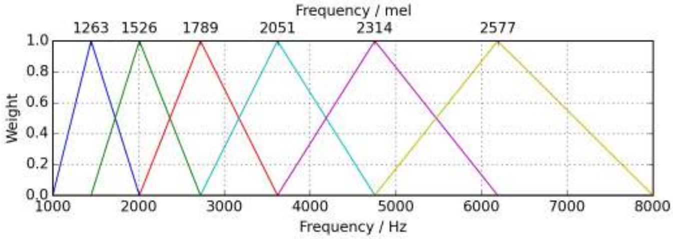


- **Fast Fourier Transform (FFT)** The Fast Fourier Transform (FFT) is applied to each windowed frame to convert it from the time domain to the frequency domain. The FFT gives us a complex-valued representation that includes both magnitude and phase information. However, for MFCCs, only the magnitude is used to calculate the Power Spectrum. The power spectrum is the square of the magnitude spectrum and measures the energy present at each frequency component.

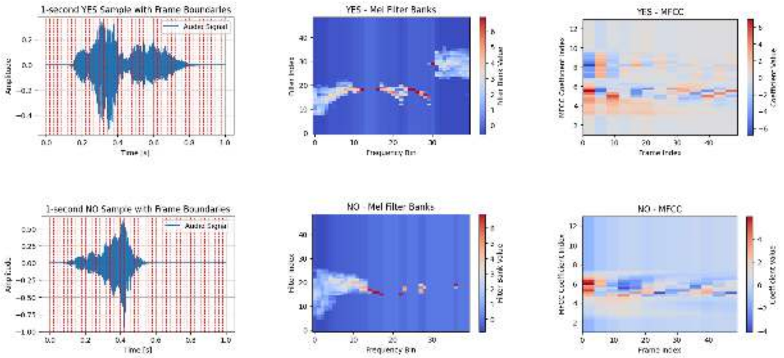
  The power spectrum $P(f)$ of a signal $x(t)$ is defined as $P(f) = |X(f)|^2$, where $X(f)$ is the Fourier Transform of $x(t)$. By squaring the magnitude of the Fourier Transform, we emphasize *stronger* frequencies over *weaker* ones, thereby capturing more relevant spectral characteristics of the audio signal. This is important in applications like audio classification, speech recognition, and Keyword Spotting (KWS), where the focus is on identifying distinct frequency patterns that characterize different classes of audio or phonemes in speech.



- **Mel Filter Banks**: The frequency domain is then mapped to the Mel scale, which approximates the human ear's response to different frequencies. The idea is to extract more features (more filter banks) in the lower frequencies and less in the high frequencies. Thus, it performs well on sounds distinguished by the human ear. Typically, 20 to 40 triangular filters extract the Mel-frequency energies. These energies are then log-transformed to convert multiplicative factors into additive ones, making them more suitable for further processing.



- **Discrete Cosine Transform (DCT)**: The last step is to apply the Discrete Cosine Transform (DCT) to the log Mel energies. The DCT helps to decorrelate the energies, effectively compressing the data and retaining only the most discriminative features. Usually, the first 12-13 DCT coefficients are retained, forming the final MFCC feature vector.



Hands-On using Python

Let's apply what we discussed while working on an actual audio sample. Open the notebook on Google CoLab and extract the MLCC features on your audio samples: [Open In Colab]

Conclusion

What Feature Extraction technique should we use?

Mel-frequency Cepstral Coefficients (MFCCs), Mel-Frequency Energies (MFEs), or Spectrogram are techniques for representing audio data, which are often helpful in different contexts.

In general, MFCCs are more focused on capturing the envelope of the power spectrum, which makes them less sensitive to fine-grained spectral details but more robust to noise. This is often desirable for speech-related tasks. On the other hand, spectrograms or MFEs preserve more detailed frequency information, which can be advantageous in tasks that require discrimination based on fine-grained spectral content.

MFCCs are particularly strong for

1. **Speech Recognition**: MFCCs are excellent for identifying phonetic content in speech signals.
2. **Speaker Identification**: They can be used to distinguish between different speakers based on voice characteristics.
3. **Emotion Recognition**: MFCCs can capture the nuanced variations in speech indicative of emotional states.
4. **Keyword Spotting**: Especially in TinyML, where low computational complexity and small feature size are crucial.

Spectrograms or MFEs are often more suitable for

1. **Music Analysis**: Spectrograms can capture harmonic and timbral structures in music, which is essential for tasks like genre classification, instrument recognition, or music transcription.
2. **Environmental Sound Classification**: In recognizing non-speech, environmental sounds (e.g., rain, wind, traffic), the full spectrogram can provide more discriminative features.
3. **Birdsong Identification**: The intricate details of bird calls are often better captured using spectrograms.
4. **Bioacoustic Signal Processing**: In applications like dolphin or bat call analysis, the fine-grained frequency information in a spectrogram can be essential.
5. **Audio Quality Assurance**: Spectrograms are often used in professional audio analysis to identify unwanted noises, clicks, or other artifacts.

Resources

- Audio_Data_Analysis Colab Notebook

DSP Spectral Features

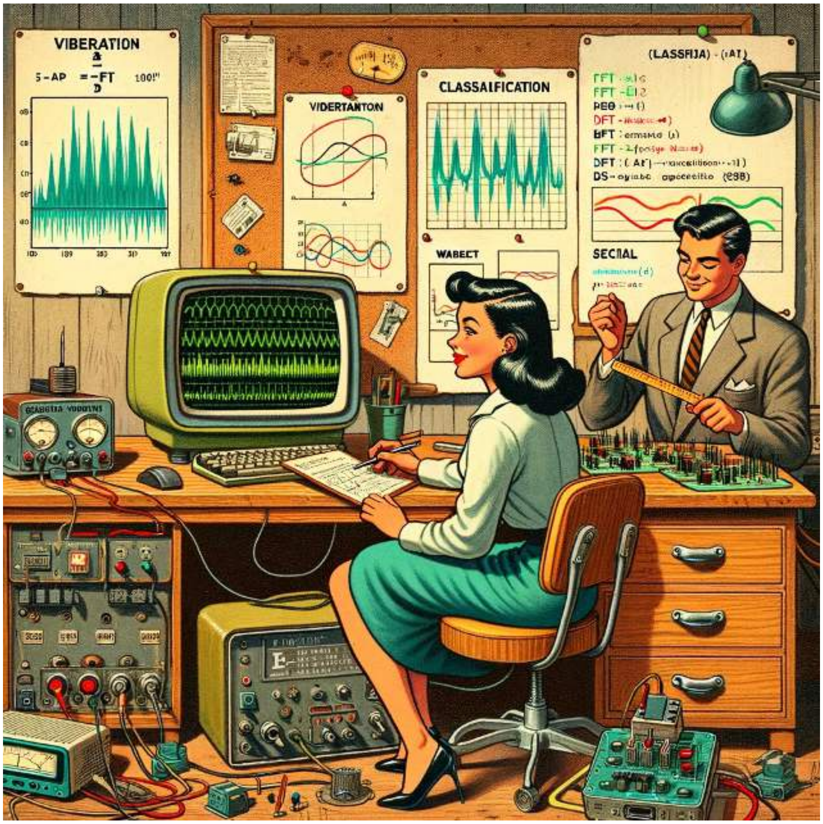


Figure 20.27: *DALL·E 3 Prompt: 1950s style cartoon illustration of a Latin male and female scientist in a vibration research room. The man is using a calculus ruler to examine ancient circuitry. The woman is at a computer with complex vibration graphs. The wooden table has boards with sensors, prominently an accelerometer. A classic, rounded-back computer shows the Arduino IDE with code for LED pin assignments and machine learning algorithms for movement detection. The Serial Monitor displays FFT, classification, wavelets, and DSPs. Vintage lamps, tools, and charts with FFT and Wavelets graphs complete the scene.*

Overview

TinyML projects related to motion (or vibration) involve data from IMUs (usually **accelerometers** and **Gyroscopes**). These time-series type datasets should be preprocessed before inputting them into a Machine Learning model training, which is a challenging area for embedded machine learning. Still, Edge Impulse helps overcome this complexity with its digital signal processing (DSP) preprocessing step and, more specifically, the Spectral Features Block for Inertial sensors.

But how does it work under the hood? Let's dig into it.

Extracting Features Review

Extracting features from a dataset captured with inertial sensors, such as accelerometers, involves processing and analyzing the raw data. Accelerometers measure the acceleration of an object along one or more axes (typically three, denoted as X, Y, and Z). These measurements can be used to understand various aspects of the object's motion, such as movement patterns and vibrations. Here's a high-level overview of the process:

Data collection: First, we need to gather data from the accelerometers. Depending on the application, data may be collected at different sampling rates. It's essential to ensure that the sampling rate is high enough to capture the relevant dynamics of the studied motion (the sampling rate should be at least double the maximum relevant frequency present in the signal).

Data preprocessing: Raw accelerometer data can be noisy and contain errors or irrelevant information. Preprocessing steps, such as filtering and normalization, can help clean and standardize the data, making it more suitable for feature extraction.

> The Studio does not perform normalization or standardization, so sometimes, when working with Sensor Fusion, it could be necessary to perform this step before uploading data to the Studio. This is particularly crucial in sensor fusion projects, as seen in this tutorial, Sensor Data Fusion with Spresense and CommonSense.

Segmentation: Depending on the nature of the data and the application, dividing the data into smaller segments or **windows** may be necessary. This can help focus on specific events or activities within the dataset, making feature extraction more manageable and meaningful. The **window size** and overlap (**window span**) choice depend on the application and the frequency of the events of interest. As a rule of thumb, we should try to capture a couple of "data cycles."

Feature extraction: Once the data is preprocessed and segmented, you can extract features that describe the motion's characteristics. Some typical features extracted from accelerometer data include:

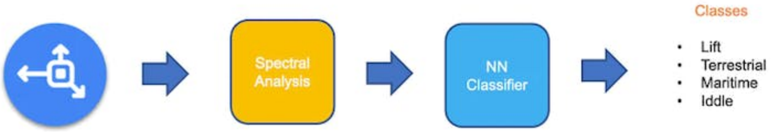
- **Time-domain** features describe the data's statistical properties within each segment, such as mean, median, standard deviation, skewness, kurtosis, and zero-crossing rate.
- **Frequency-domain** features are obtained by transforming the data into the frequency domain using techniques like the Fast Fourier Transform (FFT). Some typical frequency-domain features include the power spectrum, spectral energy, dominant frequencies (amplitude and frequency), and spectral entropy.

- **Time-frequency** domain features combine the time and frequency domain information, such as the Short-Time Fourier Transform (STFT) or the Discrete Wavelet Transform (DWT). They can provide a more detailed understanding of how the signal's frequency content changes over time.

In many cases, the number of extracted features can be large, which may lead to overfitting or increased computational complexity. Feature selection techniques, such as mutual information, correlation-based methods, or principal component analysis (PCA), can help identify the most relevant features for a given application and reduce the dimensionality of the dataset. The Studio can help with such feature-relevant calculations.

Let's explore in more detail a typical TinyML Motion Classification project covered in this series of Hands-Ons.

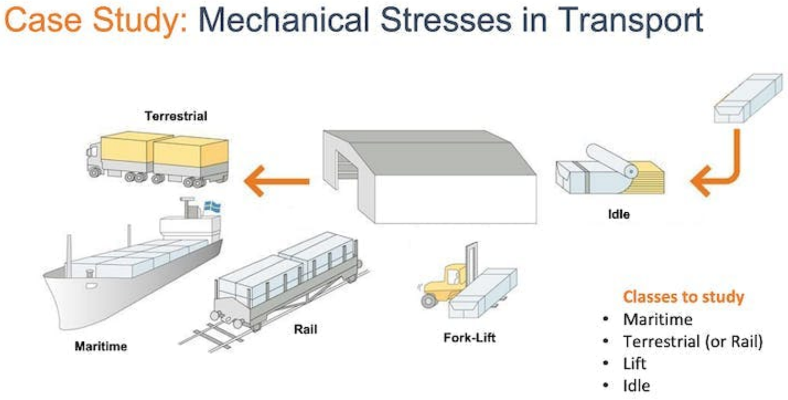
A TinyML Motion Classification project



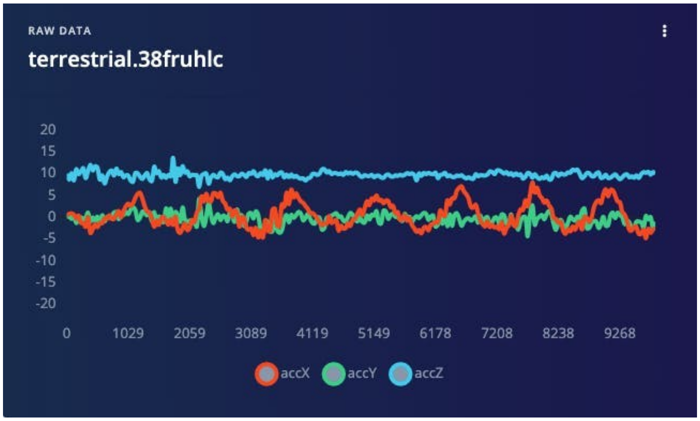
In the hands-on project, *Motion Classification and Anomaly Detection*, we simulated mechanical stresses in transport, where our problem was to classify four classes of movement:

- **Maritime** (pallets in boats)
- **Terrestrial** (pallets in a Truck or Train)
- **Lift** (pallets being handled by Fork-Lift)
- **Idle** (pallets in Storage houses)

The accelerometers provided the data on the pallet (or container).



Below is one sample (raw data) of 10 seconds, captured with a sampling frequency of 50 Hz:

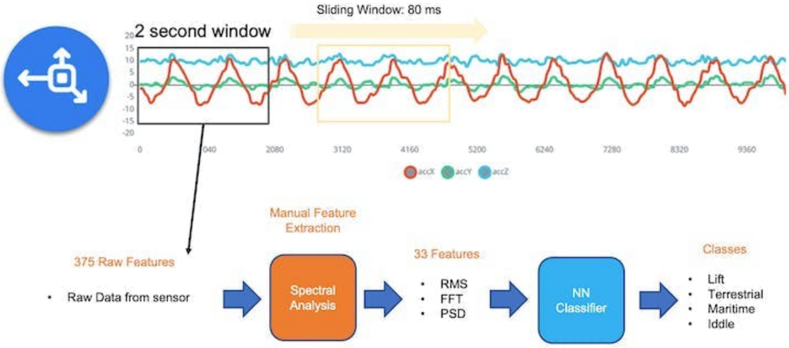


The result is similar when this analysis is done over another dataset with the same principle, using a different sampling frequency, 62.5 Hz instead of 50 Hz.

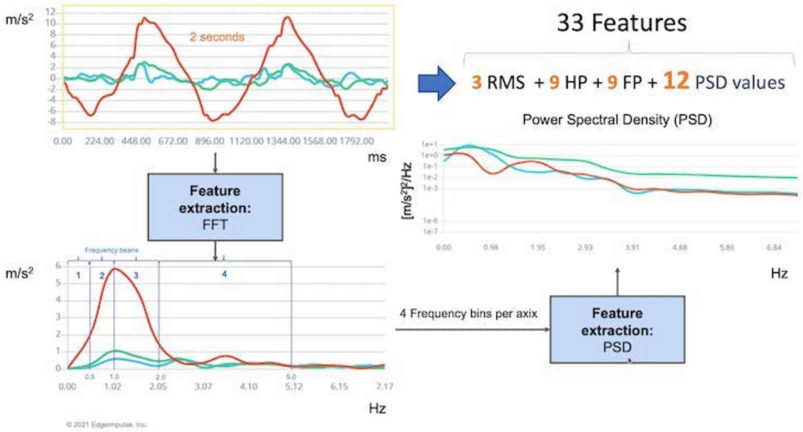
Data Pre-Processing

The raw data captured by the accelerometer (a "time series" data) should be converted to "tabular data" using one of the typical Feature Extraction methods described in the last section.

We should segment the data using a sliding window over the sample data for feature extraction. The project captured accelerometer data every 10 seconds with a sample rate of 62.5 Hz. A 2-second window captures 375 data points (3 axis × 2 seconds × 62.5 samples). The window is slid every 80 ms, creating a larger dataset where each instance has 375 "raw features."



On the Studio, the previous version (V1) of the **Spectral Analysis Block** extracted as time-domain features only the RMS, and for the frequency-domain, the peaks and frequency (using FFT) and the power characteristics (PSD) of the signal over time resulting in a fixed tabular dataset of 33 features (11 per each axis),



Those 33 features were the Input tensor of a Neural Network Classifier.

In 2022, Edge Impulse released version 2 of the Spectral Analysis block, which we will explore here.

Edge Impulse - Spectral Analysis Block V.2 under the hood

In Version 2, Time Domain Statistical features per axis/channel are:

- RMS
- Skewness
- Kurtosis

And the Frequency Domain Spectral features per axis/channel are:

- Spectral Power
- Skewness (in the next version)
- Kurtosis (in the next version)

In this link, we can have more details about the feature extraction.

> Clone the public project. You can also follow the explanation, playing with the code using my Google CoLab Notebook: Edge Impulse Spectral Analysis Block Notebook.

Start importing the libraries:

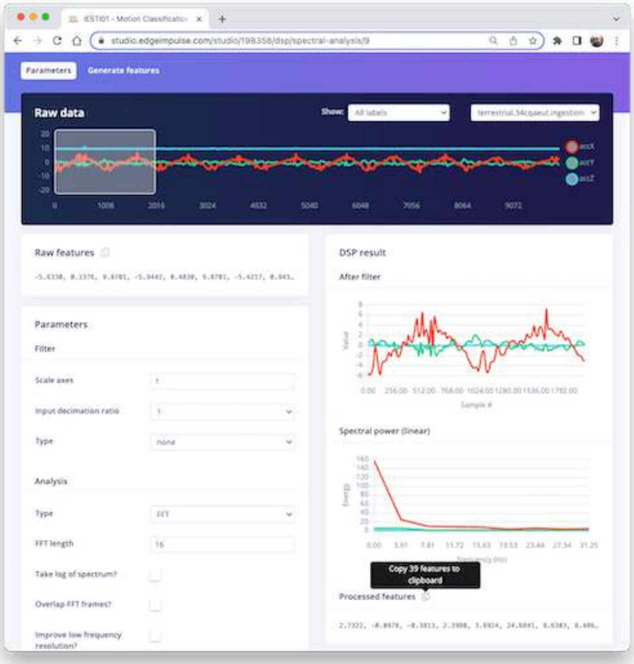
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
from scipy.stats import skew, kurtosis
from scipy import signal
from scipy.signal import welch
from scipy.stats import entropy
from sklearn import preprocessing
```

```
import pywt

plt.rcParams['figure.figsize'] = (12, 6)
plt.rcParams['lines.linewidth'] = 3
```

From the studied project, let's choose a data sample from accelerometers as below:

- Window size of 2 seconds: `[2,000]` ms
- Sample frequency: `[62.5]` Hz
- We will choose the `[None]` filter (for simplicity) and a
- FFT length: `[16]`.

```
f =  62.5 # Hertz
wind_sec = 2 # seconds
FFT_Lenght = 16
axis = ['accX', 'accY', 'accZ']
n_sensors = len(axis)
```



Raw features

-5.6330, 0.2376, 9.8701, -5.9442, 0.4830, 9.8701, -5.4217, 0.943…

Parameters

Filter

Scale axes: 1

Input decimation ratio: 1

Type: none

Analysis

Type: FFT

FFT length: 16

Take log of spectrum? □

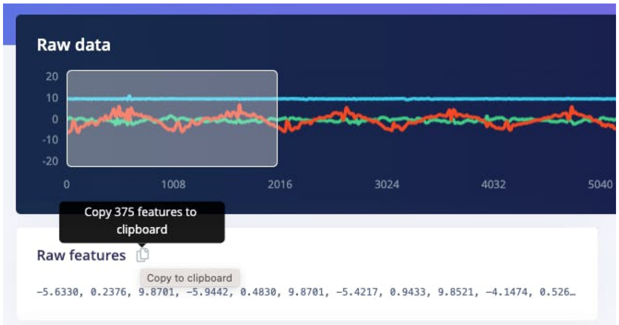
Overlap FFT frames? □

Improve low frequency resolution? □

Processed features

2.7322, -0.0978, -0.3813, 2.3980, 3.6924, 24.6845, 9.6383, 0.486…

Selecting the *Raw Features* on the Studio Spectral Analysis tab, we can copy all 375 data points of a particular 2-second window to the clipboard.



Raw features

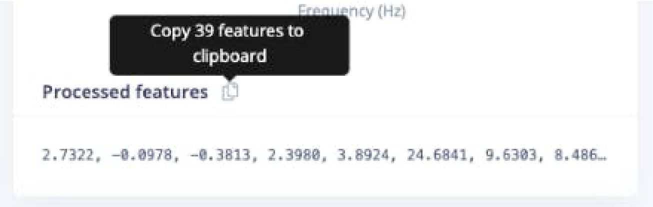
-5.6330, 0.2376, 9.8701, -5.9442, 0.4830, 9.8701, -5.4217, 0.9433, 9.8521, -4.1474, 0.526…

Paste the data points to a new variable *data*:

```
data = [
    -5.6330,  0.2376,  9.8701,
    -5.9442,  0.4830,  9.8701,
    -5.4217, ...
]
No_raw_features = len(data)
N = int(No_raw_features/n_sensors)
```

The total raw features are 375, but we will work with each axis individually, where $N = 125$ (number of samples per axis).

We aim to understand how Edge Impulse gets the processed features.



Processed features

2.7322, -0.0978, -0.3813, 2.3980, 3.8924, 24.6841, 9.6303, 8.486…

So, you should also past the processed features on a variable (to compare the calculated features in Python with the ones provided by the Studio) :

```
features = [
    2.7322, -0.0978, -0.3813,
    2.3980, 3.8924, 24.6841,
    9.6303, ...
]
N_feat = len(features)
N_feat_axis = int(N_feat/n_sensors)
```

The total number of processed features is 39, which means 13 features/axis.

Looking at those 13 features closely, we will find 3 for the time domain (RMS, Skewness, and Kurtosis):

- [rms] [skew] [kurtosis]

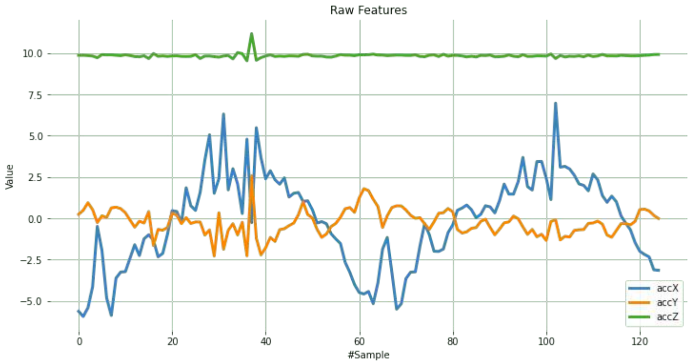
and 10 for the frequency domain (we will return to this later).

- [spectral skew][spectral kurtosis][Spectral Power 1] ... [Spectral Power 8]

Splitting raw data per sensor

The data has samples from all axes; let's split and plot them separately:

```
def plot_data(sensors, axis, title):
    [plt.plot(x, label=y) for x,y in zip(sensors, axis)]
    plt.legend(loc='lower right')
    plt.title(title)
    plt.xlabel('#Sample')
    plt.ylabel('Value')
    plt.box(False)
    plt.grid()
    plt.show()

accX = data[0::3]
accY = data[1::3]
accZ = data[2::3]
sensors = [accX, accY, accZ]
plot_data(sensors, axis, 'Raw Features')
```



Subtracting the mean

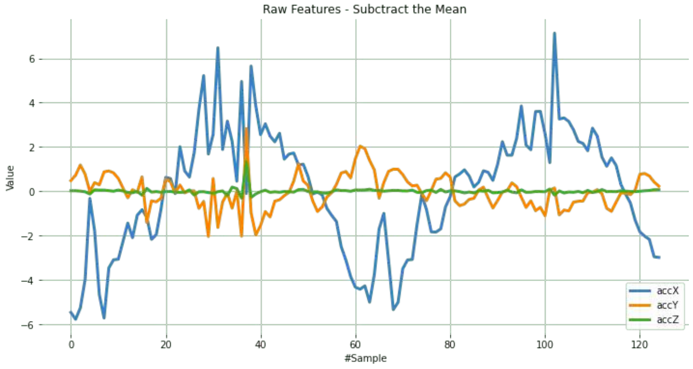
Next, we should subtract the mean from the *data*. Subtracting the mean from a data set is a common data pre-processing step in statistics and machine learning. The purpose of subtracting the mean from the data is to center the data around zero. This is important because it can reveal patterns and relationships that might be hidden if the data is not centered.

Here are some specific reasons why subtracting the mean can be helpful:

- It simplifies analysis: By centering the data, the mean becomes zero, making some calculations simpler and easier to interpret.

- It removes bias: If the data is biased, subtracting the mean can remove it and allow for a more accurate analysis.
- It can reveal patterns: Centering the data can help uncover patterns that might be hidden if the data is not centered. For example, centering the data can help you identify trends over time if you analyze a time series dataset.
- It can improve performance: In some machine learning algorithms, centering the data can improve performance by reducing the influence of outliers and making the data more easily comparable. Overall, subtracting the mean is a simple but powerful technique that can be used to improve the analysis and interpretation of data.

```
dtmean = [
    (sum(x) / len(x))
    for x in sensors
]

[
    print('mean_' + x + ' =', round(y, 4))
    for x, y in zip(axis, dtmean)
][0]

accX = [(x - dtmean[0]) for x in accX]
accY = [(x - dtmean[1]) for x in accY]
accZ = [(x - dtmean[2]) for x in accZ]
sensors = [accX, accY, accZ]

plot_data(sensors, axis, 'Raw Features - Subctract the Mean')
```



Time Domain Statistical features

RMS Calculation

The RMS value of a set of values (or a continuous-time waveform) is the square root of the arithmetic mean of the squares of the values or the square of the function that defines the continuous waveform. In physics, the RMS value of an electrical current is defined as the “value of the direct current that dissipates the same power in a resistor.”

In the case of a set of n values $x_1, x_2, \ldots, x_n$, the RMS is:

$$x_{\text{RMS}} = \sqrt{\frac{1}{n}\left(x_1^2 + x_2^2 + \cdots + x_n^2\right)}$$

> NOTE that the RMS value is different for the original raw data, and after subtracting the mean

```
# Using numpy and standardized data (subtracting mean)
rms = [np.sqrt(np.mean(np.square(x))) for x in sensors]
```

We can compare the calculated RMS values here with the ones presented by Edge Impulse:

```
[print('rms_'+x+'= ', round(y, 4)) for x,y in zip(axis, rms)][0]
print("\nCompare with Edge Impulse result features")
print(features[0:N_feat:N_feat_axis])
```

```
rms_accX=  2.7322
rms_accY=  0.7833
rms_accZ=  0.1383
```

Compared with Edge Impulse result features:

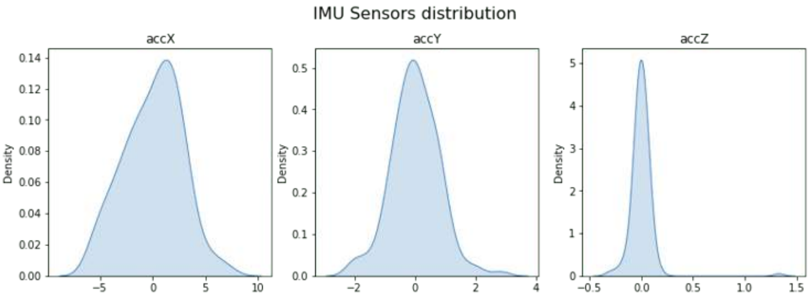
```
[2.7322, 0.7833, 0.1383]
```

Skewness and kurtosis calculation

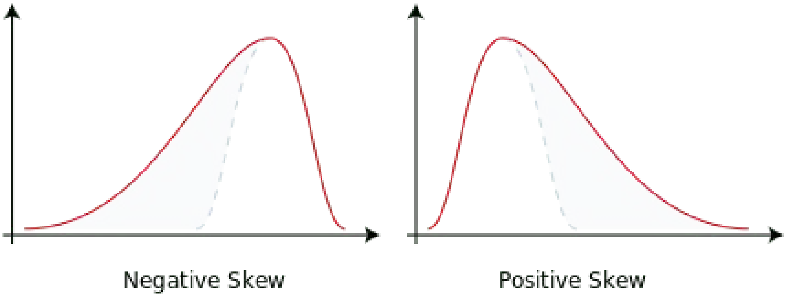
In statistics, skewness and kurtosis are two ways to measure the **shape of a distribution**.

Here, we can see the sensor values distribution:

```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(13, 4))
sns.kdeplot(accX, fill=True, ax=axes[0])
sns.kdeplot(accY, fill=True, ax=axes[1])
sns.kdeplot(accZ, fill=True, ax=axes[2])
axes[0].set_title('accX')
axes[1].set_title('accY')
axes[2].set_title('accZ')
plt.suptitle('IMU Sensors distribution', fontsize=16, y=1.02)
plt.show()
```



Skewness is a measure of the asymmetry of a distribution. This value can be positive or negative.

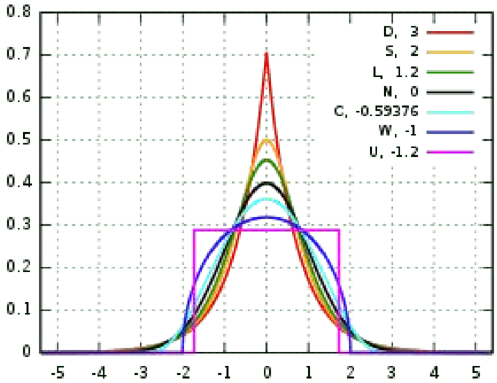


- A negative skew indicates that the tail is on the left side of the distribution, which extends towards more negative values.
- A positive skew indicates that the tail is on the right side of the distribution, which extends towards more positive values.
- A zero value indicates no skewness in the distribution at all, meaning the distribution is perfectly symmetrical.

```
skew = [skew(x, bias=False) for x in sensors]
[print('skew_'+x+'= ', round(y, 4))
  for x,y in zip(axis, skew)][0]
print("\nCompare with Edge Impulse result features")
features[1:N_feat:N_feat_axis]
```

```
skew_accX=  -0.099
skew_accY=  0.1756
skew_accZ=  6.9463
Compared with Edge Impulse result features:
[-0.0978, 0.1735, 6.8629]
```

Kurtosis is a measure of whether or not a distribution is heavy-tailed or light-tailed relative to a normal distribution.



- The kurtosis of a normal distribution is zero.
- If a given distribution has a negative kurtosis, it is said to be playkurtic, which means it tends to produce fewer and less extreme outliers than the normal distribution.
- If a given distribution has a positive kurtosis , it is said to be leptokurtic, which means it tends to produce more outliers than the normal distribution.

```
kurt = [kurtosis(x, bias=False) for x in sensors]
[print('kurt_'+x+'= ', round(y, 4))
  for x,y in zip(axis, kurt)][0]
print("\nCompare with Edge Impulse result features")
features[2:N_feat:N_feat_axis]
```

```
kurt_accX=  -0.3475
kurt_accY=  1.2673
kurt_accZ=  68.1123
Compared with Edge Impulse result features:
[-0.3813, 1.1696, 65.3726]
```

Spectral features

The filtered signal is passed to the Spectral power section, which computes the **FFT** to generate the spectral features.

Since the sampled window is usually larger than the FFT size, the window will be broken into frames (or "sub-windows"), and the FFT is calculated over each frame.

FFT length - The FFT size. This determines the number of FFT bins and the resolution of frequency peaks that can be separated. A low number means more signals will average together in the same FFT bin, but it also reduces the number of features and model size. A high number will separate more signals into separate bins, generating a larger model.

- The total number of Spectral Power features will vary depending on how you set the filter and FFT parameters. With No filtering, the number of features is 1/2 of the FFT Length.

Spectral Power - Welch's method

We should use Welch's method to split the signal on the frequency domain in bins and calculate the power spectrum for each bin. This method divides the signal into overlapping segments, applies a window function to each segment, computes the periodogram of each segment using DFT, and averages them to obtain a smoother estimate of the power spectrum.

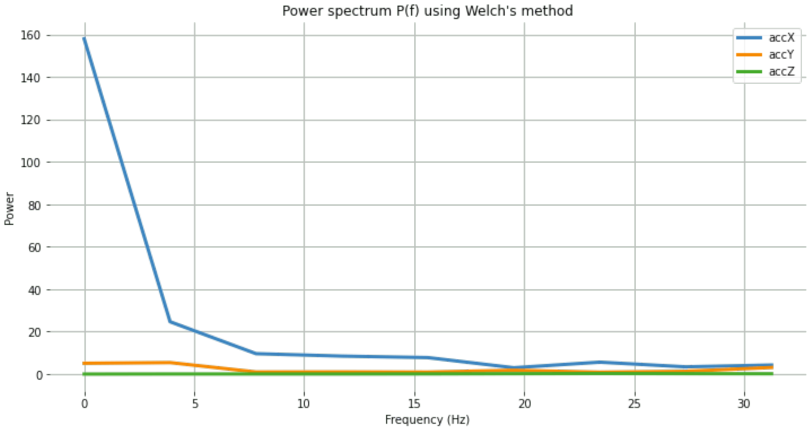
```
# Function used by Edge Impulse instead of scipy.signal.welch().
def welch_max_hold(fx, sampling_freq, nfft, n_overlap):
    n_overlap = int(n_overlap)
    spec_powers = [0 for _ in range(nfft//2+1)]
    ix = 0
    while ix <= len(fx):
        # Slicing truncates if end_idx > len,
        # and rfft will auto-zero pad
        fft_out = np.abs(np.fft.rfft(fx[ix:ix+nfft], nfft))
        spec_powers = np.maximum(spec_powers, fft_out**2/nfft)
        ix = ix + (nfft-n_overlap)
    return np.fft.rfftfreq(nfft, 1/sampling_freq), spec_powers
```

Applying the above function to 3 signals:

```
fax,Pax = welch_max_hold(accX, fs, FFT_Lenght, 0)
fay,Pay = welch_max_hold(accY, fs, FFT_Lenght, 0)
faz,Paz = welch_max_hold(accZ, fs, FFT_Lenght, 0)
specs = [Pax, Pay, Paz ]
```

We can plot the Power Spectrum P(f):

```
plt.plot(fax,Pax, label='accX')
plt.plot(fay,Pay, label='accY')
plt.plot(faz,Paz, label='accZ')
plt.legend(loc='upper right')
plt.xlabel('Frequency (Hz)')
#plt.ylabel('PSD [V**2/Hz]')
plt.ylabel('Power')
plt.title('Power spectrum P(f) using Welch's method')
plt.grid()
plt.box(False)
plt.show()
```



Besides the Power Spectrum, we can also include the skewness and kurtosis of the features in the frequency domain (should be available on a new version):

```
spec_skew = [skew(x, bias=False) for x in specs]
spec_kurtosis = [kurtosis(x, bias=False) for x in specs]
```

Let's now list all Spectral features per axis and compare them with EI:

```
print("EI Processed Spectral features (accX): ")
print(features[3:N_feat_axis][0:])
print("\nCalculated features:")
print (round(spec_skew[0],4))
print (round(spec_kurtosis[0],4))
[print(round(x, 4)) for x in Pax[1:]][0]
```

```
EI Processed Spectral features (accX):
2.398, 3.8924, 24.6841, 9.6303, 8.4867, 7.7793, 2.9963, 5.6242, 3.4198, 4.2735
Calculated features:
2.9069 8.5569 24.6844 9.6304 8.4865 7.7794 2.9964 5.6242 3.4198 4.2736
```

```
print("EI Processed Spectral features (accY): ")
print(features[16:26][0:]) # 13: 3+N_feat_axis;
                           # 26 = 2x N_feat_axis
print("\nCalculated features:")
print (round(spec_skew[1],4))
print (round(spec_kurtosis[1],4))
[print(round(x, 4)) for x in Pay[1:]][0]
```

```
EI Processed Spectral features (accY):
0.9426, -0.8039, 5.429, 0.999, 1.0315, 0.9459, 1.8117, 0.9088, 1.3302, 3.112
Calculated features:
1.1426 -0.3886 5.4289 0.999 1.0315 0.9458 1.8116 0.9088 1.3301 3.1121
```

```
print("EI Processed Spectral features (accZ): ")
print(features[29:][0:]) #29: 3+(2*N_feat_axis);
print("\nCalculated features:")
print (round(spec_skew[2],4))
print (round(spec_kurtosis[2],4))
[print(round(x, 4)) for x in Paz[1:]][0]
```

```
EI Processed Spectral features (accZ):

0.3117, -1.3812, 0.0606, 0.057, 0.0567, 0.0976, 0.194, 0.2574, 0.2083, 0.166

Calculated features:

0.3781 -1.4874 0.0606 0.057 0.0567 0.0976 0.194 0.2574 0.2083 0.166
```

Time-frequency domain

Wavelets

Wavelet is a powerful technique for analyzing signals with transient features or abrupt changes, such as spikes or edges, which are difficult to interpret with traditional Fourier-based methods.

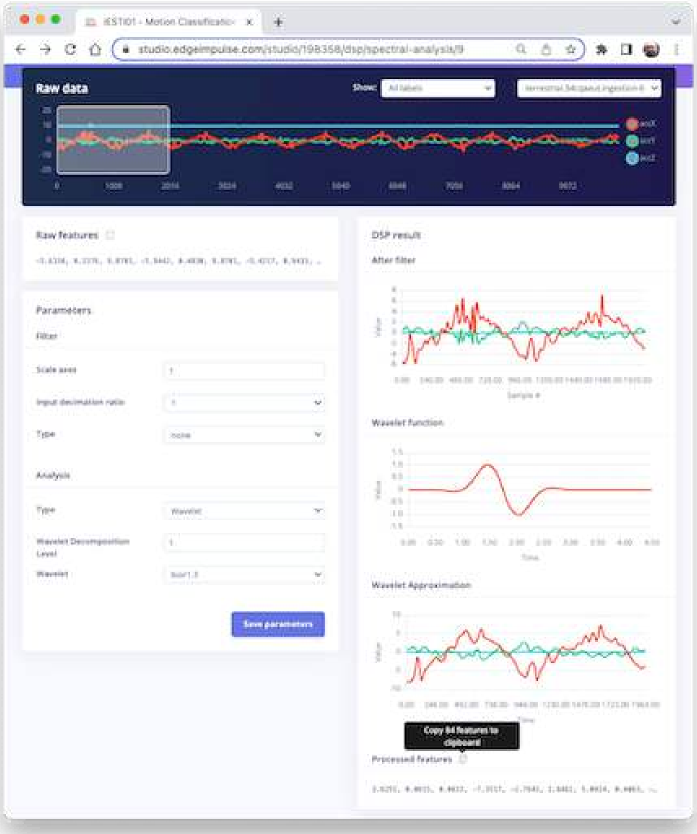
Wavelet transforms work by breaking down a signal into different frequency components and analyzing them individually. The transformation is achieved by convolving the signal with a **wavelet function**, a small waveform centered at a specific time and frequency. This process effectively decomposes the signal into different frequency bands, each of which can be analyzed separately.

One of the critical benefits of wavelet transforms is that they allow for time-frequency analysis, which means that they can reveal the frequency content of a signal as it changes over time. This makes them particularly useful for analyzing non-stationary signals, which vary over time.

Wavelets have many practical applications, including signal and image compression, denoising, feature extraction, and image processing.

Let's select Wavelet on the Spectral Features block in the same project:

- Type: Wavelet
- Wavelet Decomposition Level: 1
- Wavelet: bior1.3



Raw features

-1.6338, 4.2378, 3.8781, -1.5442, 4.4038, 3.8781, -1.4217, 8.9431, ...

Parameters

Filter

Scale axes: 1

Input decimation ratio: 1

Type: none

Analysis

Type: Wavelet

Wavelet Decomposition Level: 1

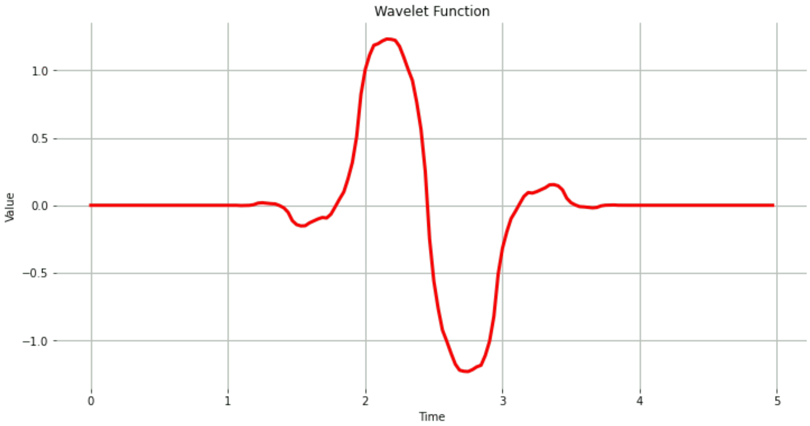
Wavelet: bior1.3

Processed features

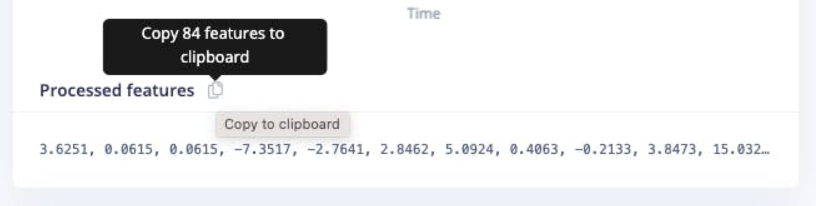
3.0231, 4.0015, 4.0017, -7.3517, -2.7541, 1.4481, 5.4414, 0.4461, ...

The Wavelet Function

```
wavelet_name='bior1.3'
num_layer = 1

wavelet = pywt.Wavelet(wavelet_name)
[phi_d,psi_d,phi_r,psi_r,x] = wavelet.wavefun(level=5)
plt.plot(x, psi_d, color='red')
plt.title('Wavelet Function')
plt.ylabel('Value')
plt.xlabel('Time')
plt.grid()
plt.box(False)
plt.show()
```



As we did before, let's copy and past the Processed Features:



```
3.6251, 0.0615, 0.0615, -7.3517, -2.7641, 2.8462, 5.0924, 0.4063, -0.2133, 3.8473, 15.032…
```

```
features = [
    3.6251, 0.0615, 0.0615,
    -7.3517, -2.7641, 2.8462,
    5.0924, ...
]
N_feat = len(features)
N_feat_axis = int(N_feat/n_sensors)
```

Edge Impulse computes the Discrete Wavelet Transform (DWT) for each one of the Wavelet Decomposition levels selected. After that, the features will be extracted.

In the case of **Wavelets**, the extracted features are *basic statistical values*, *crossing values*, and *entropy.* There are, in total, 14 features per layer as below:

- [11] Statiscal Features: **n5, n25, n75, n95, mean, median,** standard deviation **(std)**, variance **(var)** root mean square **(rms), kurtosis**, and skewness **(skew)**.
- [2] Crossing Features: Zero crossing rate **(zcross)** and mean crossing rate **(mcross)** are the times that the signal passes through the baseline $(y = 0)$ and the average level (y = u) per unit of time, respectively
- [1] Complexity Feature: **Entropy** is a characteristic measure of the complexity of the signal

All the above 14 values are calculated for each Layer (including L0, the original signal)

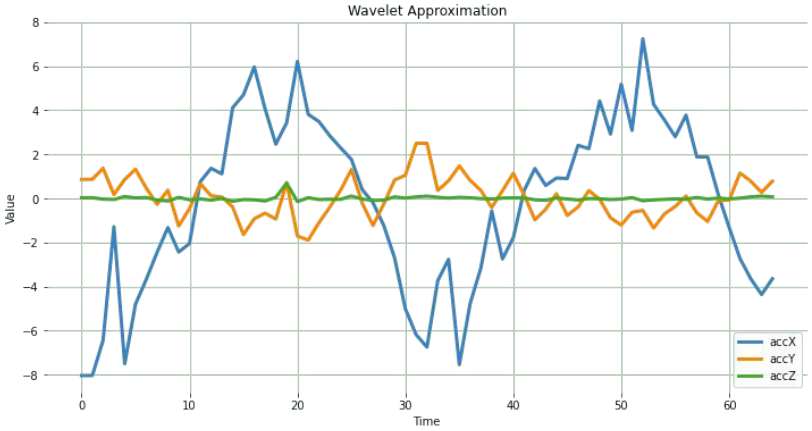
- The total number of features varies depending on how you set the filter and the number of layers. For example, with [None] filtering and Level[1], the number of features per axis will be 14×2 (L0 and L1) = 28. For the three axes, we will have a total of 84 features.

Wavelet Analysis

Wavelet analysis decomposes the signal (**accX, accY, and accZ**) into different frequency components using a set of filters, which separate these components into low-frequency (slowly varying parts of the signal containing long-term patterns), such as **accX_l1, accY_l1, accZ_l1** and, high-frequency (rapidly varying parts of the signal containing short-term patterns) components, such as **accX_d1, accY_d1, accZ_d1**, permitting the extraction of features for further analysis or classification.

Only the low-frequency components (approximation coefficients, or cA) will be used. In this example, we assume only one level (Single-level Discrete Wavelet Transform), where the function will return a tuple. With a multilevel decomposition, the "Multilevel 1D Discrete Wavelet Transform", the result will be a list (for detail, please see: Discrete Wavelet Transform (DWT))

```
(accX_l1, accX_d1) = pywt.dwt(accX, wavelet_name)
(accY_l1, accY_d1) = pywt.dwt(accY, wavelet_name)
(accZ_l1, accZ_d1) = pywt.dwt(accZ, wavelet_name)
sensors_l1 = [accX_l1, accY_l1, accZ_l1]

# Plot power spectrum versus frequency
plt.plot(accX_l1, label='accX')
plt.plot(accY_l1, label='accY')
plt.plot(accZ_l1, label='accZ')
plt.legend(loc='lower right')
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Wavelet Approximation')
plt.grid()
plt.box(False)
plt.show()
```



Feature Extraction

Let's start with the basic statistical features. Note that we apply the function for both the original signals and the resultant cAs from the DWT:

```
def calculate_statistics(signal):
    n5 = np.percentile(signal, 5)
    n25 = np.percentile(signal, 25)
    n75 = np.percentile(signal, 75)
    n95 = np.percentile(signal, 95)
    median = np.percentile(signal, 50)
    mean = np.mean(signal)
    std = np.std(signal)
    var = np.var(signal)
    rms = np.sqrt(np.mean(np.square(signal)))
    return [n5, n25, n75, n95, median, mean, std, var, rms]

stat_feat_l0 = [calculate_statistics(x) for x in sensors]
stat_feat_l1 = [calculate_statistics(x) for x in sensors_l1]
```

The Skelness and Kurtosis:

```
skew_l0 = [skew(x, bias=False) for x in sensors]
skew_l1 = [skew(x, bias=False) for x in sensors_l1]
kurtosis_l0 = [kurtosis(x, bias=False) for x in sensors]
kurtosis_l1 = [kurtosis(x, bias=False) for x in sensors_l1]
```

Zero crossing (zcross) is the number of times the wavelet coefficient crosses the zero axis. It can be used to measure the signal's frequency content since high-frequency signals tend to have more zero crossings than low-frequency signals.

Mean crossing (mcross), on the other hand, is the number of times the wavelet coefficient crosses the mean of the signal. It can be used to measure the amplitude since high-amplitude signals tend to have more mean crossings than low-amplitude signals.

```
def getZeroCrossingRate(arr):
    my_array = np.array(arr)
    zcross = float(
        "{:.2f}".format(
          (((my_array[:-1] * my_array[1:]) < 0).sum()) / len(arr)
        )
    )
    return zcross


def getMeanCrossingRate(arr):
    mcross = getZeroCrossingRate(np.array(arr) - np.mean(arr))
    return mcross

def calculate_crossings(list):
    zcross=[]
    mcross=[]
    for i in range(len(list)):
        zcross_i = getZeroCrossingRate(list[i])
        zcross.append(zcross_i)
        mcross_i = getMeanCrossingRate(list[i])
        mcross.append(mcross_i)
    return zcross, mcross

cross_l0 = calculate_crossings(sensors)
cross_l1 = calculate_crossings(sensors_l1)
```

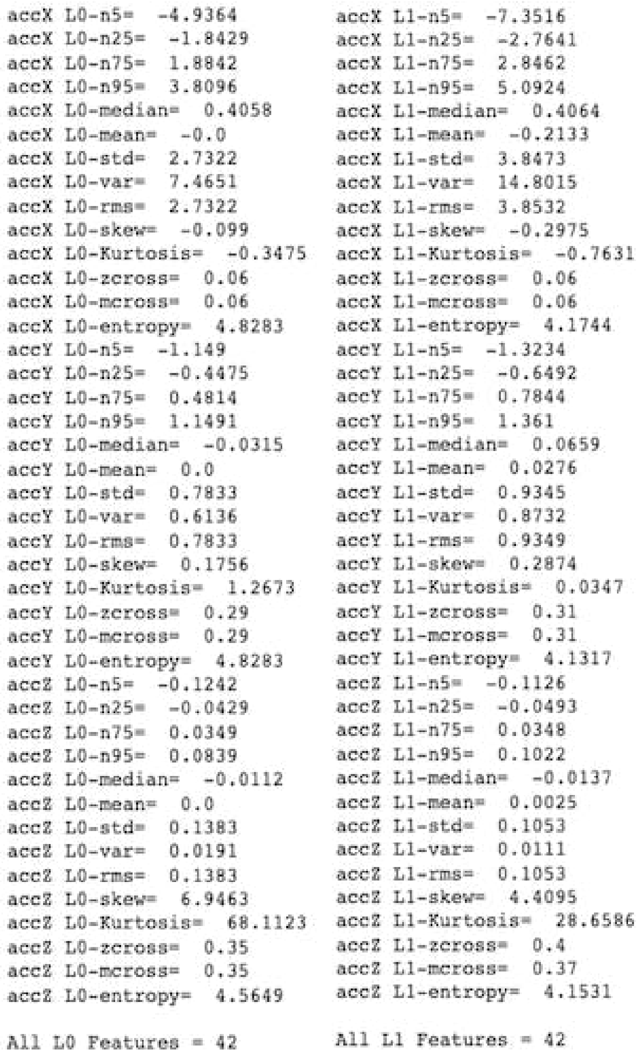
In wavelet analysis, **entropy** refers to the degree of disorder or randomness in the distribution of wavelet coefficients. Here, we used Shannon entropy, which measures a signal's uncertainty or randomness. It is calculated as the negative sum of the probabilities of the different possible outcomes of the signal multiplied by their base 2 logarithm. In the context of wavelet analysis, Shannon entropy can be used to measure the complexity of the signal, with higher values indicating greater complexity.

```
def calculate_entropy(signal, base=None):
    value, counts = np.unique(signal, return_counts=True)
    return entropy(counts, base=base)

entropy_l0 = [calculate_entropy(x) for x in sensors]
entropy_l1 = [calculate_entropy(x) for x in sensors_l1]
```

Let's now list all the wavelet features and create a list by layers.

```
L1_features_names = [
    "L1-n5", "L1-n25", "L1-n75", "L1-n95", "L1-median",
    "L1-mean", "L1-std", "L1-var", "L1-rms", "L1-skew",
    "L1-Kurtosis", "L1-zcross", "L1-mcross", "L1-entropy"
]

L0_features_names = [
    "L0-n5", "L0-n25", "L0-n75", "L0-n95", "L0-median",
    "L0-mean", "L0-std", "L0-var", "L0-rms", "L0-skew",
    "L0-Kurtosis", "L0-zcross", "L0-mcross", "L0-entropy"
]

all_feat_l0 = []
for i in range(len(axis)):
    feat_l0 = (
        stat_feat_l0[i]
        + [skew_l0[i]]
        + [kurtosis_l0[i]]
        + [cross_l0[0][i]]
        + [cross_l0[1][i]]
        + [entropy_l0[i]]
    )
    [print(axis[i] + ' +x+= ', round(y, 4))
       for x, y in zip(L0_features_names, feat_l0)][0]
    all_feat_l0.append(feat_l0)

all_feat_l0 = [
    item
    for sublist in all_feat_l0
    for item in sublist
]
print(f"\nAll L0 Features = {len(all_feat_l0)}")

all_feat_l1 = []
for i in range(len(axis)):
    feat_l1 = (
        stat_feat_l1[i]
        + [skew_l1[i]]
        + [kurtosis_l1[i]]
        + [cross_l1[0][i]]
        + [cross_l1[1][i]]
        + [entropy_l1[i]]
    )
    [print(axis[i]+' '+x+'= ', round(y, 4))
       for x,y in zip(L1_features_names, feat_l1)][0]
    all_feat_l1.append(feat_l1)
```

```
all_feat_l1 = [
    item
    for sublist in all_feat_l1
    for item in sublist
]
print(f"\nAll L1 Features = {len(all_feat_l1)}")
```



```
accX L0-n5=  -4.9364          accX L1-n5=  -7.3516
accX L0-n25=  -1.8429         accX L1-n25=  -2.7641
accX L0-n75=  1.8842          accX L1-n75=  2.8462
accX L0-n95=  3.8096          accX L1-n95=  5.0924
accX L0-median=  0.4058       accX L1-median=  0.4064
accX L0-mean=  -0.0           accX L1-mean=  -0.2133
accX L0-std=  2.7322          accX L1-std=  3.8473
accX L0-var=  7.4651          accX L1-var=  14.8015
accX L0-rms=  2.7322          accX L1-rms=  3.8532
accX L0-skew=  -0.099         accX L1-skew=  -0.2975
accX L0-Kurtosis=  -0.3475    accX L1-Kurtosis=  -0.7631
accX L0-zcross=  0.06         accX L1-zcross=  0.06
accX L0-mcross=  0.06         accX L1-mcross=  0.06
accX L0-entropy=  4.8283      accX L1-entropy=  4.1744
accY L0-n5=  -1.149           accY L1-n5=  -1.3234
accY L0-n25=  -0.4475         accY L1-n25=  -0.6492
accY L0-n75=  0.4814          accY L1-n75=  0.7844
accY L0-n95=  1.1491          accY L1-n95=  1.361
accY L0-median=  -0.0315      accY L1-median=  0.0659
accY L0-mean=  0.0            accY L1-mean=  0.0276
accY L0-std=  0.7833          accY L1-std=  0.9345
accY L0-var=  0.6136          accY L1-var=  0.8732
accY L0-rms=  0.7833          accY L1-rms=  0.9349
accY L0-skew=  0.1756         accY L1-skew=  0.2874
accY L0-Kurtosis=  1.2673     accY L1-Kurtosis=  0.0347
accY L0-zcross=  0.29         accY L1-zcross=  0.31
accY L0-mcross=  0.29         accY L1-mcross=  0.31
accY L0-entropy=  4.8283      accY L1-entropy=  4.1317
accZ L0-n5=  -0.1242          accZ L1-n5=  -0.1126
accZ L0-n25=  -0.0429         accZ L1-n25=  -0.0493
accZ L0-n75=  0.0349          accZ L1-n75=  0.0348
accZ L0-n95=  0.0839          accZ L1-n95=  0.1022
accZ L0-median=  -0.0112      accZ L1-median=  -0.0137
accZ L0-mean=  0.0            accZ L1-mean=  0.0025
accZ L0-std=  0.1383          accZ L1-std=  0.1053
accZ L0-var=  0.0191          accZ L1-var=  0.0111
accZ L0-rms=  0.1383          accZ L1-rms=  0.1053
accZ L0-skew=  6.9463         accZ L1-skew=  4.4095
accZ L0-Kurtosis=  68.1123    accZ L1-Kurtosis=  28.6586
accZ L0-zcross=  0.35         accZ L1-zcross=  0.4
accZ L0-mcross=  0.35         accZ L1-mcross=  0.37
accZ L0-entropy=  4.5649      accZ L1-entropy=  4.1531

All L0 Features = 42          All L1 Features = 42
```

Conclusion

Edge Impulse Studio is a powerful online platform that can handle the preprocessing task for us. Still, given our engineering perspective, we want to understand what is happening under the hood. This knowledge will help us find the best options and hyper-parameters for tuning our projects.

Daniel Situnayake wrote in his blog: "Raw sensor data is highly dimensional and noisy. Digital signal processing algorithms help us sift the signal from the noise. DSP is an essential part of embedded engineering, and many edge processors have on-board acceleration for DSP. As an ML engineer, learning basic DSP gives you superpowers for handling high-frequency time series data in your models." I recommend you read Dan's excellent post in its totality: nn to cpp: What you need to know about porting deep learning models to the edge.

APPENDIX

PhD Survival Guide

Technical knowledge in machine learning systems or be it in any other field, while essential, is only one dimension of successful research and scholarship. The journey through (graduate) school and beyond demands a broader set of skills: the ability to read and synthesize complex literature, communicate ideas effectively, manage time, and navigate academic careers thoughtfully.

This appendix is a small set of resources that address these important but often underdiscussed aspects of academic life. The curated materials span from seminal works that have guided multiple generations of researchers to contemporary discussions of productivity and scientific communication.

Many of these resources originated in computer science and engineering contexts, with each section focusing on a distinct aspect of academic life and presenting authoritative sources that have proven particularly valuable for graduate students and early-career researchers.

If you have suggestions or recommendations, please feel free to contact me vj[@]eecs harvard edu or issue a GitHub PR with your suggestion!

Career Advice

On Research Careers and Productivity

1. How to Have a Bad Career in Research/Academia A humorous and insightful guide by Turing Award winner David Patterson on common pitfalls to avoid in academic research.
2. You and Your Research A famous lecture by Richard Hamming on how to do impactful research and why some researchers excel.
3. Ten Simple Rules for Doing Your Best Research, According to Hamming A summary and expansion on Richard Hamming's principles, providing practical and motivational guidance for researchers at all stages.
4. The Importance of Stupidity in Scientific Research A short essay by Martin A. Schwartz on embracing the challenges of research and learning to thrive in the unknown.
5. Advice to a Young Scientist A classic book by Peter Medawar offering practical and philosophical advice on scientific research careers.

On Reading and Learning

1. How to Read a Paper A guide by S. Keshav on how to efficiently read and understand research papers.
2. Efficient Reading of Papers in Science and Technology Practical advice by Michael J. Hanson for handling the large volume of research papers in technical fields.

On Time Management and Productivity

1. Deep Work By Cal Newport, this book provides strategies for focusing deeply and maximizing productivity in cognitively demanding tasks.
2. Applying to Ph.D. Programs in Computer Science) A guide by Mor Harchol-Balter on time management, research strategies, and thriving during a Ph.D.
3. The Unwritten Laws of Engineering Though focused on engineering, W. J. King offers timeless advice on professionalism and effectiveness in technical work.

On Oral Presentation Advice

1. Oral Presentation Advice A concise guide by Mark Hill on delivering clear and engaging oral presentations in academic and technical contexts.
2. How to Give a Good Research Talk A guide by Simon Peyton Jones, John Hughes, and John Launchbury on crafting and delivering effective research presentations.
3. Ten Simple Rules for Making Good Oral Presentations A practical set of tips published by PLOS Computational Biology for delivering impactful oral presentations.

On Writing and Communicating Science

Any suggestions?

Video Resources

1. **You and Your Research by Richard Hamming** A video lecture of Richard Hamming's talk on achieving significant research contributions.
2. **How to Write a Great Research Paper** Simon Peyton Jones shares tips on writing research papers and presenting ideas effectively.

REFERENCES

References

0001, Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, et al. 2018a. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 578–94. https://www.usenix.org/conference/osdi18/presentation/chen.

———, et al. 2018b. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." In *OSDI*, 578–94. https://www.usenix.org/conference/osdi18/presentation/chen.

0003, Mu Li, David G. Andersen, Alexander J. Smola, and Kai Yu. 2014. "Communication Efficient Distributed Machine Learning with the Parameter Server." In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, edited by Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, 19–27. https://proceedings.neurips.cc/paper/2014/hash/1ff1de774005f8da13f42943881c655f-Abstract.html.

Abadi, Martin, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. "Deep Learning with Differential Privacy." In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 308–18. CCS '16. New York, NY, USA: ACM. https://doi.org/10.1145/2976749.2978318.

Abadi, Martín, Ashish Agarwal, Paul Barham, et al. 2015. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." Google Brain.

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2016. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." *arXiv Preprint arXiv:1603.04467*, March. http://arxiv.org/abs/1603.04467v2.

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. "TensorFlow: A System for Large-Scale Machine Learning." In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 265–83. USENIX Association. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Abdelkader, Ahmed, Michael J. Curry, Liam Fowl, Tom Goldstein, Avi Schwarzschild, Manli Shu, Christoph Studer, and Chen Zhu. 2020. "Headless Horseman: Adversarial Attacks on Transfer Learning Models." In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing*

(ICASSP), 3087–91. IEEE. https://doi.org/10.1109/icassp40776.2020.9053181.

Abdelkhalik, Hamdy, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A. Badawy. 2022. "Demystifying the Nvidia Ampere Architecture Through Microbenchmarking and Instruction-Level Analysis." In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. https://doi.org/10.1109/hpec55821.2022.9926299.

Addepalli, Sravanti, B. S. Vivek, Arya Baburaj, Gaurang Sriramanan, and R. Venkatesh Babu. 2020. "Towards Achieving Adversarial Robustness by Enforcing Feature Consistency Across Bit Planes." In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 1020–29. IEEE. https://doi.org/10.1109/cvpr42600.2020.00110.

Adi, Yossi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. 2018. "Turning Your Weakness into a Strength: Watermarking Deep Neural Networks by Backdooring." In *27th USENIX Security Symposium (USENIX Security 18)*, 1615–31.

Agrawal, Dakshi, Selcuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. 2007. "Trojan Detection Using IC Fingerprinting." In *2007 IEEE Symposium on Security and Privacy (SP '07)*, 296–310. Springer; IEEE. https://doi.org/10.1109/sp.2007.36.

Ahmadilivani, Mohammad Hasan, Mahdi Taheri, Jaan Raik, Masoud Daneshtalab, and Maksim Jenihhin. 2024. "A Systematic Literature Review on Hardware Reliability Assessment Methods for Deep Neural Networks." *ACM Computing Surveys* 56 (6): 1–39. https://doi.org/10.1145/3638242.

Ahmed, Reyan, Greg Bodwin, Keaton Hamm, Stephen Kobourov, and Richard Spence. 2021. "On Additive Spanners in Weighted Graphs with Local Error." *arXiv Preprint arXiv:2103.09731* 64 (12): 58–65. https://doi.org/10.1145/3467017.

Akidau, Tyler, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, et al. 2015. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing." *Proceedings of the VLDB Endowment* 8 (12): 1792–1803. https://doi.org/10.14778/2824032.2824076.

Altayeb, Moez, Marco Zennaro, and Marcelo Rovai. 2022. "Classifying Mosquito Wingbeat Sound Using TinyML." In *Proceedings of the 2022 ACM Conference on Information Technology for Social Good*, 132–37. ACM. https://doi.org/10.1145/3524458.3547258.

Alvim, Mário S., Konstantinos Chatzikokolakis, Yusuke Kawamoto, and Catuscia Palamidessi. 2022. "Information Leakage Games: Exploring Information as a Utility Function." *ACM Transactions on Privacy and Security* 25 (3): 1–36. https://doi.org/10.1145/3517330.

Amershi, Saleema, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. "Software Engineering for Machine Learning: A Case Study." In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 291–300. IEEE. https://doi.org/10.1109/icse-seip.2019.00042.

Amiel, Frederic, Christophe Clavier, and Michael Tunstall. 2006. "Fault Analysis of DPA-Resistant Algorithms." In *Fault Diagnosis and Tolerance in Cryptography*, 223–36. Springer; Springer Berlin Heidelberg. https://doi.org/10.1007/11889700/_20.

Amodei, Dario, Danny Hernandez, et al. 2018. "AI and Compute." *OpenAI Blog*. https://openai.com/research/ai-and-compute.

Amodei, Dario, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. "Concrete Problems in AI Safety." *arXiv Preprint arXiv:1606.06565*, June. http://arxiv.org/abs/1606.06565v2.

Andrae, Anders, and Tomas Edler. 2015. "On Global Electricity Usage of Communication Technology: Trends to 2030." *Challenges* 6 (1): 117–57. https://doi.org/10.3390/challe6010117.

Antonakakis, Manos, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, et al. 2017. "Understanding the Mirai Botnet." In *26th USENIX Security Symposium (USENIX Security 17)*, 16:1093–1110.

Ardila, Rosana, Megan Branson, Kelly Davis, Michael Kohler, Josh Meyer, Michael Henretty, Reuben Morais, Lindsay Saunders, Francis Tyers, and Gregor Weber. 2020. "Common Voice: A Massively-Multilingual Speech Corpus." In *Proceedings of the Twelfth Language Resources and Evaluation Conference*, 4218–22. Marseille, France: European Language Resources Association. https://aclanthology.org/2020.lrec-1.520.

Arifeen, Tooba, Abdus Sami Hassan, and Jeong-A Lee. 2020. "Approximate Triple Modular Redundancy: A Survey." *IEEE Access* 8: 139851–67. https://doi.org/10.1109/access.2020.3012673.

Arivazhagan, Manoj Ghuhan, Vinay Aggarwal, Aaditya Kumar Singh, and Sunav Choudhary. 2019. "Federated Learning with Personalization Layers." *CoRR* abs/1912.00818 (December). http://arxiv.org/abs/1912.00818v1.

Asonov, D., and R. Agrawal. n.d. "Keyboard Acoustic Emanations." In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, 3–11. IEEE; IEEE. https://doi.org/10.1109/secpri.2004.1301311.

Ateniese, Giuseppe, Luigi V. Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. 2015. "Hacking Smart Machines with Smarter Ones: How to Extract Meaningful Data from Machine Learning Classifiers." *International Journal of Security and Networks* 10 (3): 137. https://doi.org/10.1504/ijsn.2015.071829.

Attia, Zachi I., Alan Sugrue, Samuel J. Asirvatham, Michael J. Ackerman, Suraj Kapa, Paul A. Friedman, and Peter A. Noseworthy. 2018. "Noninvasive Assessment of Dofetilide Plasma Concentration Using a Deep Learning (Neural Network) Analysis of the Surface Electrocardiogram: A Proof of Concept Study." *PLOS ONE* 13 (8): e0201059. https://doi.org/10.1371/journal.pone.0201059.

Aygun, Sercan, Ece Olcay Gunes, and Christophe De Vleeschouwer. 2021. "Efficient and Robust Bitstream Processing in Binarised Neural Networks." *Electronics Letters* 57 (5): 219–22. https://doi.org/10.1049/ell2.12045.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. "Layer Normalization." *arXiv Preprint arXiv:1607.06450*, July. http://arxiv.org/abs/1607.06450v1.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2014. "Neural Machine Translation by Jointly Learning to Align and Translate." *arXiv Preprint arXiv:1409.0473*, September. http://arxiv.org/abs/1409.0473v7.

Bai, Tao, Jinqi Luo, Jun Zhao, Bihan Wen, and Qian Wang. 2021. "Recent Advances in Adversarial Training for Adversarial Robustness." *arXiv Preprint arXiv:2102.01356*, February. http://arxiv.org/abs/2102.01356v5.

Bamoumen, Hatim, Anas Temouden, Nabil Benamar, and Yousra Chtouki. 2022. "How TinyML Can Be Leveraged to Solve Environmental Problems: A Survey." In *2022 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, 338–43. IEEE; IEEE. https://doi.org/10.1109/3ict56508.2022.9990661.

Banbury, Colby R., Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, et al. 2020. "Benchmarking TinyML Systems: Challenges and Direction." *arXiv Preprint arXiv:2003.04821*, March. http://arxiv.org/abs/2003.04821v4.

Banbury, Colby, Emil Njor, Andrea Mattia Garavagno, Matthew Stewart, Pete Warden, Manjunath Kudlur, Nat Jeffries, Xenofon Fafoutis, and Vijay Janapa Reddi. 2024. "Wake Vision: A Tailored Dataset and Benchmark Suite for TinyML Computer Vision Applications," May. http://arxiv.org/abs/2405.00892v4.

Banbury, Colby, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, et al. 2021. "MLPerf Tiny Benchmark." *arXiv Preprint arXiv:2106.07597*, June. http://arxiv.org/abs/2106.07597v4.

Bannon, Pete, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. 2019. "Computer and Redundancy Solution for the Full Self-Driving Computer." In *2019 IEEE Hot Chips 31 Symposium (HCS)*, 1–22. IEEE Computer Society; IEEE. https://doi.org/10.1109/hotchips.2019.8875645.

Baraglia, David, and Hokuto Konno. 2019. "On the Bauer-Furuta and Seiberg-Witten Invariants of Families of 4-Manifolds." *arXiv Preprint arXiv:1903.01649*, March, 8955–67. http://arxiv.org/abs/1903.01649v3.

Bardenet, Rémi, Olivier Cappé, Gersende Fort, and Balázs Kégl. 2015. "Adaptive MCMC with Online Relabeling." *Bernoulli* 21 (3). https://doi.org/10.3150/13-bej578.

Barenghi, Alessandro, Guido M. Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. 2010. "Low Voltage Fault Attacks to AES." In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 7–12. IEEE; IEEE. https://doi.org/10.1109/hst.2010.5513121.

Barocas, Solon, Moritz Hardt, and Arvind Narayanan. 2023. *Fairness and Machine Learning: Limitations and Opportunities*. MIT Press.

Barroso, Luiz André, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Springer International Publishing. https://doi.org/10.1007/978-3-031-01741-4.

Barroso, Luiz André, and Urs Hölzle. 2007b. "The Case for Energy-Proportional Computing." *Computer* 40 (12): 33–37. https://doi.org/10.1109/mc.2007.443.

———. 2007a. "The Case for Energy-Proportional Computing." *Computer* 40 (12): 33–37. https://doi.org/10.1109/mc.2007.443.

Barroso, Luiz André, Urs Hölzle, and Parthasarathy Ranganathan. 2019. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Springer International Publishing. https://doi.org/10.1007/978-3-031-01761-2.

Baydin, Atilim Gunes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017a. "Automatic Differentiation in Machine Learning: A Survey." *J. Mach. Learn. Res.* 18: 153:1–43. https://jmlr.org/papers/v18/17-468.html.

———. 2017b. "Automatic Differentiation in Machine Learning: A Survey." *J. Mach. Learn. Res.* 18 (153): 153:1–43. https://jmlr.org/papers/v18/17-468.html.

Beaton, Albert E., and John W. Tukey. 1974. "The Fitting of Power Series, Meaning Polynomials, Illustrated on Band-Spectroscopic Data." *Technometrics* 16 (2): 147. https://doi.org/10.2307/1267936.

Bedford Taylor, Michael. 2017. "The Evolution of Bitcoin Hardware." *Computer* 50 (9): 58–66. https://doi.org/10.1109/mc.2017.3571056.

Bender, Emily M., Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? ." In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–23. ACM. https://doi.org/10.1145/3442188.3445922.

Bengio, Emmanuel, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. 2015. "Conditional Computation in Neural Networks for Faster Models." *arXiv Preprint arXiv:1511.06297*, November. http://arxiv.org/abs/1511.06297v2.

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. 2013b. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation." *arXiv Preprint*, August. http://arxiv.org/abs/1308.3432v1.

———. 2013a. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation." *arXiv Preprint arXiv:1308.3432*, August. http://arxiv.org/abs/1308.3432v1.

Ben-Nun, Tal, and Torsten Hoefler. 2019. "Demystifying Parallel and Distributed Deep Learning: An in-Depth Concurrency Analysis." *ACM Computing Surveys* 52 (4): 1–43. https://doi.org/10.1145/3320060.

Berger, Vance W., and YanYan Zhou. 2014. "Wiley StatsRef: Statistics Reference Online." *Wiley Statsref: Statistics Reference Online*. Wiley. https://doi.org/10.1002/9781118445112.stat06558.

Bergstra, James, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. "Theano: A CPU and GPU Math Compiler in Python." In *Proceedings of the 9th Python in Science Conference*, 4:18–24. 1. SciPy. https://doi.org/10.25080/majora-92bf1922-003.

Beyer, Lucas, Olivier J. Hénaff, Alexander Kolesnikov, Xiaohua Zhai, and Aäron van den Oord. 2020. "Are We Done with ImageNet?" *arXiv Preprint arXiv:2006.07159*, June. http://arxiv.org/abs/2006.07159v1.

Bhagoji, Arjun Nitin, Warren He, Bo Li, and Dawn Song. 2018. "Practical Black-Box Attacks on Deep Neural Networks Using Efficient Query Mechanisms." In *Computer Vision – ECCV 2018*, 158–74. Springer International Publishing. https://doi.org/10.1007/978-3-030-01258-8/_10.

Bhamra, Ran, Adrian Small, Christian Hicks, and Olimpia Pilch. 2024. "Impact Pathways: Geopolitics, Risk and Ethics in Critical Minerals Supply Chains." *International Journal of Operations &Amp; Production Management*, September. https://doi.org/10.1108/ijopm-03-2024-0228.

Biggio, Battista, Blaine Nelson, and Pavel Laskov. 2012. "Poisoning Attacks Against Support Vector Machines." In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress. http://icml.cc/2012/papers/880.pdf.

Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, et al. 2011. "The Gem5 Simulator." *ACM SIGARCH Computer Architecture News* 39 (2): 1–7. https://doi.org/10.1145/2024716.2024718.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. Springer.

Blackwood, Jayden, Frances C. Wright, Nicole J. Look Hong, and Anna R. Gagliardi. 2019. "Quality of DCIS Information on the Internet: A Content Analysis." *Breast Cancer Research and Treatment* 177 (2): 295–305. https://doi.org/10.1007/s10549-019-05315-8.

Bolchini, Cristiana, Luca Cassano, Antonio Miele, and Alessandro Toschi. 2023. "Fast and Accurate Error Simulation for CNNs Against Soft Errors." *IEEE Transactions on Computers* 72 (4): 984–97. https://doi.org/10.1109/tc.2022.3184274.

Bommasani, Rishi, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, et al. 2021. "On the Opportunities and Risks of Foundation Models." *arXiv Preprint arXiv:2108.07258*, August. http://arxiv.org/abs/2108.07258v3.

Bouri, Elie. 2015. "A Broadened Causality in Variance Approach to Assess the Risk Dynamics Between Crude Oil Prices and the Jordanian Stock Market." *Energy Policy* 85 (October): 271–79. https://doi.org/10.1016/j.enpol.2015.06.001.

Bourtoule, Lucas, Varun Chandrasekaran, Christopher A. Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. 2021. "Machine Unlearning." In *2021 IEEE Symposium on Security and Privacy (SP)*, 141–59. IEEE; IEEE. https://doi.org/10.1109/sp40001.2021.00019.

Bradbury, James, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, et al. 2018. "JAX: Composable Transformations of Python+NumPy Programs." http://github.com/google/jax.

Brain, Google. 2020. "XLA: Optimizing Compiler for Machine Learning." *TensorFlow Blog*. https://tensorflow.org/xla.

———. 2022. *TensorFlow Documentation*. https://www.tensorflow.org/.

Brakerski, Zvika et al. 2022. "Federated Learning and the Rise of Edge Intelligence: Challenges and Opportunities." *Communications of the ACM* 65 (8): 54–63.

Breck, Eric, Shanqing Cai, Eric Nielsen, Mohamed Salib, and D. Sculley. 2020. "The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction." *IEEE Transactions on Big Data* 6 (2): 347–61.

Breier, Jakub, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. 2018. "DeepLaser: Practical Fault Attack on Deep Neural Networks." *ArXiv*

Preprint abs/1806.05859 (June): 619–33. http://arxiv.org/abs/1806.05859v2.

Brown, Samantha. 2021. "Long-Term Software Support: A Key Factor in Sustainable AI Hardware." *Computer Ethics and Sustainability* 14 (2): 112–30.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, and et al. 2020. "Language Models Are Few-Shot Learners." *Advances in Neural Information Processing Systems (NeurIPS)* 33: 1877–1901.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. "Language Models Are Few-Shot Learners." *arXiv Preprint arXiv:2005.14165*, May. http://arxiv.org/abs/2005.14165v4.

Brynjolfsson, Erik, and Andrew McAfee. 2014. *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies, 1st Edition.* W. W. Norton Company.

Buolamwini, Joy, and Timnit Gebru. 2018. "Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification." In *Conference on Fairness, Accountability and Transparency*, 77–91. PMLR. http://proceedings.mlr.press/v81/buolamwini18a.html.

Burnet, David, and Richard Thomas. 1989. "Spycatcher: The Commodification of Truth." *Journal of Law and Society* 16 (2): 210. https://doi.org/10.2307/1410360.

Bursztein, Elie, Luca Invernizzi, Karel Král, Daniel Moghimi, Jean-Michel Picod, and Marina Zhang. 2024. "Generalized Power Attacks Against Crypto Hardware Using Long-Range Deep Learning." *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024 (3): 472–99. https://doi.org/10.46586/tches.v2024.i3.472-499.

Bursztein, Elie, Luca Invernizzi, Karel Král, and Jean-Michel Picod. 2019. "SCAAML: Side Channel Attacks Assisted with Machine Learning." https://github.com/google/scaaml.

Bushnell, Michael L, and Vishwani D Agrawal. 2002. "Built-in Self-Test." *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, 489–548.

C. P. Baldé, V. Gray, V. Forti. 2017. "The Global e-Waste Monitor 2017: Quantities, Flows and Resources." *United Nations University, International Telecommunication Union, International Solid Waste Association*.https://www.itu.int/en/ITU-D/Climate-Change/Documents/GEM\%202017/Global-E-waste\%20Monitor\%202017\%20.pdf .

Cai, Carrie J., Emily Reif, Narayan Hegde, Jason Hipp, Been Kim, Daniel Smilkov, Martin Wattenberg, et al. 2019. "Human-Centered Tools for Coping with Imperfect Algorithms During Medical Decision-Making." In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, edited by Jennifer G. Dy and Andreas Krause, 80:1–14. Proceedings of Machine Learning Research. ACM. https://doi.org/10.1145/3290605.3300234.

Cai, Han, Chuang Gan, and Song Han. 2020. "Once-for-All: Train One Network and Specialize It for Efficient Deployment." In *International Conference on Learning Representations*.

Calvo, Rafael A., Dorian Peters, Karina Vold, and Richard M. Ryan. 2020. "Supporting Human Autonomy in AI Systems: A Framework for Ethical

Enquiry." In *Ethics of Digital Well-Being*, 31–54. Springer International Publishing. https://doi.org/10.1007/978-3-030-50585-1_2.

Carlini, Nicholas, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang 0001, Micah Sherr, Clay Shields, David A. Wagner 0001, and Wenchao Zhou. 2016. "Hidden Voice Commands." In *25th USENIX Security Symposium (USENIX Security 16)*, 513–30. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/carlini.

Carlini, Nicholas, Daniel Paleka, Krishnamurthy Dj Dvijotham, Thomas Steinke, Jonathan Hayase, A. Feder Cooper, Katherine Lee, et al. 2024. "Stealing Part of a Production Language Model." *arXiv Preprint arXiv:2403.06634*, March. http://arxiv.org/abs/2403.06634v2.

Chandola, Varun, Arindam Banerjee, and Vipin Kumar. 2009. "Anomaly Detection: A Survey." *ACM Computing Surveys* 41 (3): 1–58. https://doi.org/10.1145/1541880.1541882.

Chapelle, O., B. Scholkopf, and A. Zien Eds. 2009. "Semi-Supervised Learning (Chapelle, o. Et Al., Eds.; 2006) [Book Reviews]." *IEEE Transactions on Neural Networks* 20 (3): 542–42. https://doi.org/10.1109/tnn.2009.2015974.

Chen, Chaofan, Oscar Li, Daniel Tao, Alina Barnett, Cynthia Rudin, and Jonathan Su. 2019. "This Looks Like That: Deep Learning for Interpretable Image Recognition." In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, edited by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, 8928–39. https://proceedings.neurips.cc/paper/2019/hash/adf7ee2dcf142b0e11888e72b43fcb75-Abstract.html.

Chen, Emma, Shvetank Prakash, Vijay Janapa Reddi, David Kim, and Pranav Rajpurkar. 2023. "A Framework for Integrating Artificial Intelligence for Clinical Care with Continuous Therapeutic Monitoring." *Nature Biomedical Engineering*, November. https://doi.org/10.1038/s41551-023-01115-0.

Chen, H.-W. 2006. "Gallium, Indium, and Arsenic Pollution of Groundwater from a Semiconductor Manufacturing Area of Taiwan." *Bulletin of Environmental Contamination and Toxicology* 77 (2): 289–96. https://doi.org/10.1007/s00128-006-1062-3.

Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, et al. 2021. "Evaluating Large Language Models Trained on Code." *arXiv Preprint arXiv:2107.03374*, July. http://arxiv.org/abs/2107.03374v2.

Chen, Mia Xu, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, et al. 2018. "The Best of Both Worlds: Combining Recent Advances in Neural Machine Translation." In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 30:5998–6008. Association for Computational Linguistics. https://doi.org/10.18653/v1/p18-1008.

Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems." *arXiv Preprint arXiv:1512.01274*, December. http://arxiv.org/abs/1512.01274v1.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. "Training Deep Nets with Sublinear Memory Cost." *CoRR* abs/1604.06174 (April). http://arxiv.org/abs/1604.06174v2.

Chen, Wei-Yu, Yen-Cheng Liu, Zsolt Kira, Yu-Chiang Frank Wang, and Jia-Bin Huang. 2019. "A Closer Look at Few-Shot Classification." In *International Conference on Learning Representations (ICLR)*.

Chen, Yu-Hsin, Joel Emer, and Vivienne Sze. 2017. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks." *IEEE Micro*, 1–1. https://doi.org/10.1109/mm.2017.265085944.

Chen, Yu-Hsin, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2016. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks." *IEEE Journal of Solid-State Circuits* 51 (1): 186–98. https://doi.org/10.1109/JSSC.2015.2488709.

Chen, Zitao, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2019. "<I>BinFI</i>: An Efficient Fault Injector for Safety-Critical Machine Learning Systems." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–23. SC '19. New York, NY, USA: ACM. https://doi.org/10.1145/3295500.3356177.

Chen, Zitao, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2020. "TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications." In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 426–35. IEEE; IEEE. https://doi.org/10.1109/issre5003.2020.00047.

Cheng, Eric, Shahrzad Mirkhani, Lukasz G. Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R. Stan, et al. 2016. "CLEAR: <U>c</u> Ross <u>-l</u> Ayer <u>e</u> Xploration for <u>a</u> Rchitecting <u>r</u> Esilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores." In *Proceedings of the 53rd Annual Design Automation Conference*, 1–6. ACM. https://doi.org/10.1145/2897937.2897996.

Cheng, Yu et al. 2022. "Memory-Efficient Deep Learning: Advances in Model Compression and Sparsification." *ACM Computing Surveys*.

Cheshire, David. 2021. "Circular Economy and Sustainable AI: Designing Out Waste in the Tech Industry." In *The Handbook to Building a Circular Economy*, 48–61. RIBA Publishing. https://doi.org/10.4324/9781003212775-8.

Chetlur, Sharan, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. "cuDNN: Efficient Primitives for Deep Learning." *arXiv Preprint arXiv:1410.0759*, October. http://arxiv.org/abs/1410.0759v3.

Cho, Kyunghyun, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches." In *Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8)*, 103–11. Association for Computational Linguistics.

Choi, Jungwook, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. "PACT: Parameterized Clipping Activation for Quantized Neural Networks." *arXiv Preprint*, May. http://arxiv.org/abs/1805.06085v2.

Choi, Sebin, and Sungmin Yoon. 2024. "GPT-Based Data-Driven Urban Building Energy Modeling (GPT-UBEM): Concept, Methodology, and Case Studies." *Energy and Buildings* 325 (December): 115042. https://doi.org/10.1016/j.enbuild.2024.115042.

Chollet, François et al. 2015. "Keras." *GitHub Repository*. https://github.com/fchollet/keras.

Chollet, François. 2018. "Introduction to Keras." *March 9th*.

Choquette, Jack. 2023. "NVIDIA Hopper H100 GPU: Scaling Performance." *IEEE Micro* 43 (3): 9–17. https://doi.org/10.1109/mm.2023.3256796.

Choudhary, Tejalal, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. 2020. "A Comprehensive Survey on Model Compression and Acceleration." *Artificial Intelligence Review* 53 (7): 5113–55. https://doi.org/10.1007/s10462-020-09816-7.

Chowdhery, Aakanksha, Anatoli Noy, Gaurav Misra, Zhuyun Dai, Quoc V. Le, and Jeff Dean. 2021. "Edge TPU: An Edge-Optimized Inference Accelerator for Deep Learning." In *International Symposium on Computer Architecture*.

Christiano, Paul F., Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. "Deep Reinforcement Learning from Human Preferences." In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, edited by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, 4299–4307. https://proceedings.neurips.cc/paper/2017/hash/d5e2c0adad503c91f91df240d0cd4e49-Abstract.html.

Chu, Grace, Okan Arikan, Gabriel Bender, Weijun Wang, Achille Brighton, Pieter-Jan Kindermans, Hanxiao Liu, Berkin Akin, Suyog Gupta, and Andrew Howard. 2021. "Discovering Multi-Hardware Mobile Models via Architecture Search." In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 3016–25. IEEE. https://doi.org/10.1109/cvprw53098.2021.00337.

Chung, Jae-Won, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. 2023. "Reducing Energy Bloat in Large Model Training." *ArXiv Preprint* abs/2312.06902 (December). http://arxiv.org/abs/2312.06902v3.

Ciez, Rebecca E., and J. F. Whitacre. 2019. "Examining Different Recycling Processes for Lithium-Ion Batteries." *Nature Sustainability* 2 (2): 148–56. https://doi.org/10.1038/s41893-019-0222-5.

Coleman, Cody, Edward Chou, Julian Katz-Samuels, Sean Culatana, Peter Bailis, Alexander C. Berg, Robert Nowak, Roshan Sumbaly, Matei Zaharia, and I. Zeki Yalniz. 2022. "Similarity Search for Efficient Active Learning and Search of Rare Concepts." *Proceedings of the AAAI Conference on Artificial Intelligence* 36 (6): 6402–10. https://doi.org/10.1609/aaai.v36i6.20591.

Commission, European. 2023. "Sustainable Digital Markets Act: Environmental Transparency in AI."

Contro, Filippo, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. 2021. "EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode." *arXiv Preprint arXiv:2103.09113*, March. http://arxiv.org/abs/2103.09113v1.

Cooper, Tom, Suzanne Fallender, Joyann Pafumi, Jon Dettling, Sebastien Humbert, and Lindsay Lessard. 2011. "A Semiconductor Company's Examination of Its Water Footprint Approach." In *Proceedings of the 2011 IEEE International Symposium on Sustainable Systems and Technology*, 1–6. IEEE; IEEE. https://doi.org/10.1109/issst.2011.5936865.

Cope, Gord. 2009. "Pure Water, Semiconductors and the Recession." *Global Water Intelligence* 10 (10).

Corporation, Intel. 2021. *oneDNN: Intel's Deep Learning Neural Network Library*. https://github.com/oneapi-src/oneDNN.

Corporation, NVIDIA. 2017. "GPU-Accelerated Machine Learning and Deep Learning." *Technical Report*.

———. 2021. *NVIDIA cuDNN: GPU Accelerated Deep Learning*. https://developer.nvidia.com/cudnn.

Corporation, Thinking Machines. 1992. *CM-5 Technical Summary*. Thinking Machines Corporation.

Costa, Tiago, Chen Shi, Kevin Tien, and Kenneth L. Shepard. 2019. "A CMOS 2D Transmit Beamformer with Integrated PZT Ultrasound Transducers for Neuromodulation." In *2019 IEEE Custom Integrated Circuits Conference (CICC)*, 1–4. IEEE. https://doi.org/10.1109/cicc.2019.8780236.

Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David. 2016. "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations." *Advances in Neural Information Processing Systems (NeurIPS)* 28: 3123–31.

Courbariaux, Matthieu, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1." *arXiv Preprint arXiv:1602.02830*, February. http://arxiv.org/abs/1602.02830v3.

Crankshaw, Daniel, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. "Clipper: A {Low-Latency} Online Prediction Serving System." In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 613–27.

Cui, Hongyi, Jiajun Li, and Peng et al. Xie. 2019. "A Survey on Machine Learning Compilers: Taxonomy, Challenges, and Future Directions." *ACM Computing Surveys* 52 (4): 1–39.

Curnow, H. J. 1976. "A Synthetic Benchmark." *The Computer Journal* 19 (1): 43–49. https://doi.org/10.1093/comjnl/19.1.43.

Cybenko, G. 1992. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals, and Systems* 5 (4): 455–55. https://doi.org/10.1007/bf02134016.

Dally, William J., Stephen W. Keckler, and David B. Kirk. 2021. "Evolution of the Graphics Processing Unit (GPU)." *IEEE Micro* 41 (6): 42–51. https://doi.org/10.1109/mm.2021.3113475.

Dao, Tri, Beidi Chen, Nimit Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. 2022. "Monarch: Expressive Structured Matrices for Efficient and Accurate Training," April. http://arxiv.org/abs/2204.00595v1.

David, Robert, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, et al. 2021. "Tensorflow Lite Micro: Embedded Machine

Learning for Tinyml Systems.” *Proceedings of Machine Learning and Systems* 3: 800–811.

Davies, Martin. 2011. “Endangered Elements: Critical Thinking.” In *Study Skills for International Postgraduates*, 111–30. Macmillan Education UK. https://doi.org/10.1007/978-0-230-34553-9/_8.

Davies, Mike et al. 2021. “Advancing Neuromorphic Computing with Sparse Networks.” *Nature Electronics*.

Dayarathna, Miyuru, Yonggang Wen, and Rui Fan. 2016. “Data Center Energy Consumption Modeling: A Survey.” *IEEE Communications Surveys &Amp; Tutorials* 18 (1): 732–94. https://doi.org/10.1109/comst.2015.2481183.

Dean, Jeff, David Patterson, and Cliff Young. 2018. “A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution.” *IEEE Micro* 38 (2): 21–29. https://doi.org/10.1109/mm.2018.112130030.

Dean, Jeffrey, and Sanjay Ghemawat. 2008. “MapReduce: Simplified Data Processing on Large Clusters.” *Communications of the ACM* 51 (1): 107–13. https://doi.org/10.1145/1327452.1327492.

Deng, Chulin, Yujun Zhang, and Yanzhi Wu. 2022. “TinyTrain: Learning to Train Compact Neural Networks on the Edge.” In *Proceedings of the 39th International Conference on Machine Learning (ICML)*.

Deng, Jia, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. “ImageNet: A Large-Scale Hierarchical Image Database.” In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–55. Ieee; IEEE. https://doi.org/10.1109/cvprw.2009.5206848.

Deng, Li. 2012. “The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web].” *IEEE Signal Processing Magazine* 29 (6): 141–42. https://doi.org/10.1109/msp.2012.2211477.

Deng, Yuzhe, Aryan Mokhtari, and Asuman Ozdaglar. 2021. “Adaptive Federated Optimization.” In *Proceedings of the 38th International Conference on Machine Learning (ICML)*.

Dettmers, Tim, and Luke Zettlemoyer. 2019. “Sparse Networks from Scratch: Faster Training Without Losing Performance.” *arXiv Preprint arXiv:1907.04840*, July. http://arxiv.org/abs/1907.04840v2.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding,” October, 4171–86. http://arxiv.org/abs/1810.04805v2.

Domingos, Pedro. 2016. “The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World.” *Choice Reviews Online* 53 (07): 53–3100. https://doi.org/10.5860/choice.194685.

Dongarra, Jack J., Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. 1988. “An Extended Set of FORTRAN Basic Linear Algebra Subprograms.” *ACM Transactions on Mathematical Software* 14 (1): 1–17. https://doi.org/10.1145/42288.42291.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. 2020. “An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale.” *International Conference on Learning Representations (ICLR)*, October. http://arxiv.org/abs/2010.11929v2.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. 2021. "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale." *International Conference on Learning Representations*.

Duarte, Javier, Nhan Tran, Ben Hawks, Christian Herwig, Jules Muhizi, Shvetank Prakash, and Vijay Janapa Reddi. 2022a. "FastML Science Benchmarks: Accelerating Real-Time Scientific Edge Machine Learning." *arXiv Preprint arXiv:2207.07958*, July. http://arxiv.org/abs/2207.07958v1.

———. 2022b. "FastML Science Benchmarks: Accelerating Real-Time Scientific Edge Machine Learning," July. http://arxiv.org/abs/2207.07958v1.

Duisterhof, Bardienus P., Shushuai Li, Javier Burgues, Vijay Janapa Reddi, and Guido C. H. E. de Croon. 2021. "Sniffy Bug: A Fully Autonomous Swarm of Gas-Seeking Nano Quadcopters in Cluttered Environments." In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9099–9106. IEEE; IEEE. https://doi.org/10.1109/iros51168.2021.9636217.

Dwork, Cynthia. n.d. "Differential Privacy: A Survey of Results." In *Theory and Applications of Models of Computation*, 1–19. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-79228-4/_1.

Dwork, Cynthia, and Aaron Roth. 2013. "The Algorithmic Foundations of Differential Privacy." *Foundations and Trends® in Theoretical Computer Science* 9 (3-4): 211–407. https://doi.org/10.1561/0400000042.

Egwutuoha, Ifeanyi P., David Levy, Bran Selic, and Shiping Chen. 2013. "A Survey of Fault Tolerance Mechanisms and Checkpoint/Restart Implementations for High Performance Computing Systems." *The Journal of Supercomputing* 65 (3): 1302–26. https://doi.org/10.1007/s11227-013-0884-0.

Eisenman, Assaf, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. "Check-n-Run: A Checkpointing System for Training Deep Learning Recommendation Models." In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 929–43. https://www.usenix.org/conference/nsdi22/presentation/eisenman.

Elman, Jeffrey L. 2002. "Finding Structure in Time." In *Cognitive Modeling*, 14:257–88. 2. The MIT Press. https://doi.org/10.7551/mitpress/1888.003.0015.

Elsen, Erich, Marat Dukhan, Trevor Gale, and Karen Simonyan. 2020. "Fast Sparse ConvNets." In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 14617–26. IEEE. https://doi.org/10.1109/cvpr42600.2020.01464.

Elsken, Thomas, Jan Hendrik Metzen, and Frank Hutter. 2019b. "Neural Architecture Search." In *Automated Machine Learning*, 63–77. Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5/_3.

———. 2019a. "Neural Architecture Search." In *Automated Machine Learning*, 20:63–77. 55. Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5/_3.

Emily Denton, Rob Fergus, Soumith Chintala. 2014. "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation." In *Advances in Neural Information Processing Systems (NeurIPS)*, 1269–77.

Everingham, Mark, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2009. "The Pascal Visual Object Classes (VOC) Challenge." *International Journal of Computer Vision* 88 (2): 303–38. https://doi.org/10.1007/s11263-009-0275-4.

Eykholt, Kevin, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2017. "Robust Physical-World Attacks on Deep Learning Models." *ArXiv Preprint* abs/1707.08945 (July). http://arxiv.org/abs/1707.08945v5.

Farwell, James P., and Rafal Rohozinski. 2011. "Stuxnet and the Future of Cyber War." *Survival* 53 (1): 23–40. https://doi.org/10.1080/00396338.2011.555586.

Fedus, William, Barret Zoph, and Noam Shazeer. 2021. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity." *Journal of Machine Learning Research*.

Fei-Fei, Li, R. Fergus, and P. Perona. n.d. "Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories." In *2004 Conference on Computer Vision and Pattern Recognition Workshop*. IEEE. https://doi.org/10.1109/cvpr.2004.383.

Feldman, Andrew, Sean Lie, Michael James, et al. 2020. "The Cerebras Wafer-Scale Engine: Opportunities and Challenges of Building an Accelerator at Wafer Scale." *IEEE Micro* 40 (2): 20–29. https://doi.org/10.1109/MM.2020.2975796.

Ferentinos, Konstantinos P. 2018. "Deep Learning Models for Plant Disease Detection and Diagnosis." *Computers and Electronics in Agriculture* 145 (February): 311–18. https://doi.org/10.1016/j.compag.2018.01.009.

Feurer, Matthias, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. "Auto-Sklearn: Efficient and Robust Automated Machine Learning." In *Automated Machine Learning*, 113–34. Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5/_6.

Finn, Chelsea, Pieter Abbeel, and Sergey Levine. 2017. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks." In *Proceedings of the 34th International Conference on Machine Learning (ICML)*.

Fisher, Lawrence D. 1981. "The 8087 Numeric Data Processor." *IEEE Computer* 14 (7): 19–29. https://doi.org/10.1109/MC.1981.1653991.

Flynn, M. J. 1966. "Very High-Speed Computing Systems." *Proceedings of the IEEE* 54 (12): 1901–9. https://doi.org/10.1109/proc.1966.5273.

Francalanza, Adrian, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. "A Foundation for Runtime Monitoring." In *Runtime Verification*, 8–29. Springer; Springer International Publishing. https://doi.org/10.1007/978-3-319-67531-2/_2.

Fredrikson, Matt, Somesh Jha, and Thomas Ristenpart. 2015. "Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures." In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 1322–33. ACM. https://doi.org/10.1145/2810103.2813677.

Friedman, Batya. 1996. "Value-Sensitive Design." *Interactions* 3 (6): 16–23. https://doi.org/10.1145/242485.242493.

Fursov, Ivan, Matvey Morozov, Nina Kaploukhaya, Elizaveta Kovtun, Rodrigo Rivera-Castro, Gleb Gusev, Dmitry Babaev, Ivan Kireev, Alexey Zaytsev, and Evgeny Burnaev. 2021. "Adversarial Attacks on Deep Models for Financial Transaction Records." In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery &Amp; Data Mining*, 2868–78. ACM. https://doi.org/10.1145/3447548.3467145.

Gale, Trevor, Erich Elsen, and Sara Hooker. 2019b. "The State of Sparsity in Deep Neural Networks." *arXiv Preprint arXiv:1902.09574*, February. http://arxiv.org/abs/1902.09574v1.

———. 2019a. "The State of Sparsity in Deep Neural Networks." *arXiv Preprint arXiv:1902.09574*, February. http://arxiv.org/abs/1902.09574v1.

Gale, Trevor, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2022. "MegaBlocks: Efficient Sparse Training with Mixture-of-Experts," November. http://arxiv.org/abs/2211.15841v1.

Gama, João, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. "A Survey on Concept Drift Adaptation." *ACM Computing Surveys* 46 (4): 1–37. https://doi.org/10.1145/2523813.

Gandolfi, Karine, Christophe Mourtel, and Francis Olivier. 2001. "Electromagnetic Analysis: Concrete Results." In *Cryptographic Hardware and Embedded Systems — CHES 2001*, 251–61. Springer; Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-44709-1/_21.

Gao, Yansong, Said F. Al-Sarawi, and Derek Abbott. 2020. "Physical Unclonable Functions." *Nature Electronics* 3 (2): 81–91. https://doi.org/10.1038/s41928-020-0372-5.

Garg, Harvinder Atwal. 2020. *Practical DataOps: Delivering Agile Data Science at Scale*. Berkeley, CA: Apress. https://doi.org/10.1007/978-1-4842-5494-3.

Gassend, Blaise, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. 2002. "Silicon Physical Random Functions." In *Proceedings of the 9th ACM Conference on Computer and Communications Security - CCS '02*, 148–60. ACM; ACM Press. https://doi.org/10.1145/586131.586132.

Gebru, Timnit, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. 2021b. "Datasheets for Datasets." *Communications of the ACM* 64 (12): 86–92. https://doi.org/10.1145/3458723.

———. 2021a. "Datasheets for Datasets." *Communications of the ACM* 64 (12): 86–92. https://doi.org/10.1145/3458723.

Geiger, Atticus, Hanson Lu, Thomas Icard, and Christopher Potts. 2021. "Causal Abstractions of Neural Networks." In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, Virtual*, edited by Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, 9574–86. https://proceedings.neurips.cc/paper/2021/hash/4f5c422f4d49a5a807eda27434231040-Abstract.html.

Gholami, Amir et al. 2021. "A Survey of Quantization Methods for Efficient Neural Network Inference." *IEEE Transactions on Neural Networks and Learning Systems* 32 (10): 4562–81. https://doi.org/10.1109/TNNLS.2021.3088493.

Gholami, Amir, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. "A Survey of Quantization Methods for Efficient

Neural Network Inference." *arXiv Preprint arXiv:2103.13630*, March. http://arxiv.org/abs/2103.13630v3.

Gholami, Amir, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. "AI and Memory Wall." *IEEE Micro* 44 (3): 33–39. https://doi.org/10.1109/mm.2024.3373763.

Gnad, Dennis R. E., Fabian Oboril, and Mehdi B. Tahoori. 2017. "Voltage Drop-Based Fault Attacks on FPGAs Using Valid Bitstreams." In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 1–7. IEEE; IEEE. https://doi.org/10.23919/fpl.2017.8056840.

Goldberg, David. 1991. "What Every Computer Scientist Should Know about Floating-Point Arithmetic." *ACM Computing Surveys* 23 (1): 5–48. https://doi.org/10.1145/103162.103163.

Golub, Gene H., and Charles F. Van Loan. 1996. *Matrix Computations*. Johns Hopkins University Press.

Goncalves, Andre, Priyadip Ray, Braden Soper, Jennifer Stevens, Linda Coyle, and Ana Paula Sales. 2020. "Generation and Evaluation of Synthetic Patient Data." *BMC Medical Research Methodology* 20 (1): 1–40. https://doi.org/10.1186/s12874-020-00977-1.

Gong, Ruihao, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. 2019. "Differentiable Soft Quantization: Bridging Full-Precision and Low-Bit Neural Networks." *arXiv Preprint arXiv:1908.05033*, August. http://arxiv.org/abs/1908.05033v1.

Goodfellow, Ian J., Aaron Courville, and Yoshua Bengio. 2013b. "Scaling up Spike-and-Slab Models for Unsupervised Feature Learning." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (8): 1902–14. https://doi.org/10.1109/tpami.2012.273.

———. 2013c. "Scaling up Spike-and-Slab Models for Unsupervised Feature Learning." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (8): 1902–14. https://doi.org/10.1109/tpami.2012.273.

———. 2013a. "Scaling up Spike-and-Slab Models for Unsupervised Feature Learning." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (8): 1902–14. https://doi.org/10.1109/tpami.2012.273.

Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. 2014. "Explaining and Harnessing Adversarial Examples." *ICLR*, December. http://arxiv.org/abs/1412.6572v3.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. "Generative Adversarial Networks." *Communications of the ACM* 63 (11): 139–44. https://doi.org/10.1145/3422622.

Google. n.d. "XLA: Optimizing Compiler for Machine Learning." <https://www.tensorflow.org/xla>.

Gordon, Mitchell, Kevin Duh, and Nicholas Andrews. 2020. "Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning." In *Proceedings of the 5th Workshop on Representation Learning for NLP*. Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.repl4nlp-1.18.

Gou, Jianping, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao. 2021. "Knowledge Distillation: A Survey." *International Journal of Computer Vision* 129 (6): 1789–819. https://doi.org/10.1007/s11263-021-01453-z.

Gräfe, Ralf, Qutub Syed Sha, Florian Geissler, and Michael Paulitsch. 2023. "Large-Scale Application of Fault Injection into PyTorch Models -an Extension to PyTorchFI for Validation Efficiency." In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-s)*, 56–62. IEEE; IEEE. https://doi.org/10.1109/dsn-s58398.2023.00025.

Graphcore. 2020. "The Colossus MK2 IPU Processor." *Graphcore Technical Paper*.

Groeneveld, Dirk, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, et al. 2024. "OLMo: Accelerating the Science of Language Models." *arXiv Preprint arXiv:2402.00838*, February. http://arxiv.org/abs/2402.00838v4.

Grossman, Elizabeth. 2007. *High Tech Trash: Digital Devices, Hidden Toxics, and Human Health*. Island press.

Gu, Ivy. 2023. "Deep Learning Model Compression (Ii) by Ivy Gu Medium." https://ivygdy.medium.com/deep-learning-model-compression-ii-546352ea9453.

Gudivada, Venkat N., Dhana Rao Rao, et al. 2017. "Data Quality Considerations for Big Data and Machine Learning: Going Beyond Data Cleaning and Transformations." *IEEE Transactions on Knowledge and Data Engineering*.

Gujarati, Arpan, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. "Serving DNNs Like Clockwork: Performance Predictability from the Bottom Up." In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 443–62. https://www.usenix.org/conference/osdi20/presentation/gujarati.

Gulshan, Varun, Lily Peng, Marc Coram, Martin C. Stumpe, Derek Wu, Arunachalam Narayanaswamy, Subhashini Venugopalan, et al. 2016. "Development and Validation of a Deep Learning Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs." *JAMA* 316 (22): 2402. https://doi.org/10.1001/jama.2016.17216.

Guo, Yutao, Hao Wang, Hui Zhang, Tong Liu, Zhaoguang Liang, Yunlong Xia, Li Yan, et al. 2019. "Mobile Photoplethysmographic Technology to Detect Atrial Fibrillation." *Journal of the American College of Cardiology* 74 (19): 2365–75. https://doi.org/10.1016/j.jacc.2019.08.019.

Gupta, Suyog, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. "Deep Learning with Limited Numerical Precision." In *International Conference on Machine Learning*, 1737–46. PMLR.

Gupta, Udit, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2022. "ACT: Designing Sustainable Computer Systems with an Architectural Carbon Modeling Tool." In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 784–99. ACM. https://doi.org/10.1145/3470496.3527408.

Gupta, Udit, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2022. "Chasing Carbon: The Elusive Environmental Footprint of Computing." *IEEE Micro* 42 (6): 68–78. https://doi.org/10.1109/MM.2022.3186575.

Hamming, R. W. 1950. "Error Detecting and Error Correcting Codes." *Bell System Technical Journal* 29 (2): 147–60. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x.

Han, Song, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. "EIE: Efficient Inference Engine on Compressed Deep Neural Network." In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 243–54. IEEE. https://doi.org/10.1109/isca.2016.30.

Han, Song, Huizi Mao, and William J. Dally. 2015. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." *arXiv Preprint arXiv:1510.00149*, October. http://arxiv.org/abs/1510.00149v5.

———. 2016. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." *International Conference on Learning Representations (ICLR)*.

Han, Song, Jeff Pool, John Tran, and William J. Dally. 2015. "Learning Both Weights and Connections for Efficient Neural Networks." *CoRR* abs/1506.02626 (June): 1135–43. http://arxiv.org/abs/1506.02626v3.

Handlin, Oscar. 1965. "Science and Technology in Popular Culture." *Daedalus-Us.*, 156–70.

Hard, Andrew, Kanishka Rao, Rajiv Mathews, Saurabh Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. "Federated Learning for Mobile Keyboard Prediction." In *International Conference on Learning Representations (ICLR)*.

Harris, Michael. 2023. "The Environmental Cost of Next-Generation AI Chips: Energy, Water, and Carbon Impacts." *Journal of Green Computing* 17 (1): 22–38.

Hayes, Tyler L., Kushal Kafle, Robik Shrestha, Manoj Acharya, and Christopher Kanan. 2020. "REMIND Your Neural Network to Prevent Catastrophic Forgetting." In *Computer Vision – ECCV 2020*, 466–83. Springer International Publishing. https://doi.org/10.1007/978-3-030-58598-3/_28.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016a. "Deep Residual Learning for Image Recognition." In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–78. IEEE. https://doi.org/10.1109/cvpr.2016.90.

———. 2016b. "Deep Residual Learning for Image Recognition." In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–78. IEEE. https://doi.org/10.1109/cvpr.2016.90.

He, Xuzhen. 2023a. "Accelerated Linear Algebra Compiler for Computationally Efficient Numerical Models: Success and Potential Area of Improvement." *PLOS ONE* 18 (2): e0282265. https://doi.org/10.1371/journal.pone.0282265.

———. 2023b. "Accelerated Linear Algebra Compiler for Computationally Efficient Numerical Models: Success and Potential Area of Improvement." *PLOS ONE* 18 (2): e0282265. https://doi.org/10.1371/journal.pone.0282265.

He, Yi, Prasanna Balaprakash, and Yanjing Li. 2020. "FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators." In *2020 53rd*

Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 270–81. IEEE; IEEE. https://doi.org/10.1109/micro50266.2020.00033.

He, Yihui, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. "AMC: AutoML for Model Compression and Acceleration on Mobile Devices." In *Computer Vision – ECCV 2018*, 815–32. Springer International Publishing. https://doi.org/10.1007/978-3-030-01234-2/_48.

He, Yi, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023. "Understanding and Mitigating Hardware Failures in Deep Learning Training Systems." In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 1–16. IEEE; ACM. https://doi.org/10.1145/3579371.3589105.

Hébert-Johnson, Úrsula, Michael P. Kim, Omer Reingold, and Guy N. Rothblum. 2018. "Multicalibration: Calibration for the (Computationally-Identifiable) Masses." In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, edited by Jennifer G. Dy and Andreas Krause, 80:1944–53. Proceedings of Machine Learning Research. PMLR. http://proceedings.mlr.press/v80/hebert-johnson18a.html.

Henderson, Peter, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020a. "Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning." *CoRR* abs/2002.05651 (248): 1–43. https://doi.org/10.48550/arxiv.2002.05651.

———. 2020b. "Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning." *Journal of Machine Learning Research* 21 (248): 1–43. http://arxiv.org/abs/2002.05651v2.

Hendrycks, Dan, and Thomas Dietterich. 2019. "Benchmarking Neural Network Robustness to Common Corruptions and Perturbations." *arXiv Preprint arXiv:1903.12261*, March. http://arxiv.org/abs/1903.12261v1.

Hennessy, John L., and David A. Patterson. 2019. "A New Golden Age for Computer Architecture." *Communications of the ACM* 62 (2): 48–60. https://doi.org/10.1145/3282307.

Hennessy, John L, and David A Patterson. 2003. "Computer Architecture: A Quantitative Approach." *Morgan Kaufmann*.

Hernandez, Danny, Tom B. Brown, et al. 2020. "Measuring the Algorithmic Efficiency of Neural Networks." *OpenAI Blog*. https://openai.com/research/ai-and-efficiency.

Hernandez, Danny, and Tom B. Brown. 2020. "Measuring the Algorithmic Efficiency of Neural Networks." *arXiv Preprint arXiv:2007.03051*, May. https://doi.org/10.48550/arxiv.2005.04305.

Hestness, Joel, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2017. "Deep Learning Scaling Is Predictable, Empirically." *arXiv Preprint arXiv:1712.00409*, December. http://arxiv.org/abs/1712.00409v1.

Himmelstein, Gracie, David Bates, and Li Zhou. 2022. "Examination of Stigmatizing Language in the Electronic Health Record." *JAMA Network Open* 5 (1): e2144967. https://doi.org/10.1001/jamanetworkopen.2021.44967.

Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. 2015a. "Distilling the Knowledge in a Neural Network." *arXiv Preprint arXiv:1503.02531*, March. http://arxiv.org/abs/1503.02531v1.

———. 2015b. "Distilling the Knowledge in a Neural Network." *arXiv Preprint arXiv:1503.02531*, March. http://arxiv.org/abs/1503.02531v1.

Hirschberg, Julia, and Christopher D. Manning. 2015. "Advances in Natural Language Processing." *Science* 349 (6245): 261–66. https://doi.org/10.1126/science.aaa8685.

Hochreiter, Sepp. 1998. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions." *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 06 (02): 107–16. https://doi.org/10.1142/s0218488598000094.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation* 9 (8): 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735.

Hoefler, Torsten, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. "Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks." *arXiv Preprint arXiv:2102.00554* 22 (January): 1–124. http://arxiv.org/abs/2102.00554v1.

Hoefler, Torsten, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandros Nikolaos Ziogas. 2021. "Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks." *Journal of Machine Learning Research* 22 (241): 1–124.

Hoffmann, Jordan, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, et al. 2022. "Training Compute-Optimal Large Language Models." *arXiv Preprint arXiv:2203.15556*, March. http://arxiv.org/abs/2203.15556v1.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. 1989. "Multilayer Feedforward Networks Are Universal Approximators." *Neural Networks* 2 (5): 359–66. https://doi.org/10.1016/0893-6080(89)90020-8.

Horowitz, Mark. 2014. "1.1 Computing's Energy Problem (and What We Can Do about It)." In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE. https://doi.org/10.1109/isscc.2014.6757323.

Hosseini, Hossein, Sreeram Kannan, Baosen Zhang, and Radha Poovendran. 2017. "Deceiving Google's Perspective API Built for Detecting Toxic Comments." *ArXiv Preprint* abs/1702.08138 (February). http://arxiv.org/abs/1702.08138v1.

Houlsby, Neil, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Chloé de Laroussilhe, Andrea Gesmundo, Mohammad Attariyan, and Sylvain Gelly. 2019. "Parameter-Efficient Transfer Learning for NLP." In *International Conference on Machine Learning*, 2790–99. PMLR.

Howard, Andrew G., Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017a. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," April. http://arxiv.org/abs/1704.04861v1.

———. 2017b. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *ArXiv Preprint* abs/1704.04861 (April). http://arxiv.org/abs/1704.04861v1.

Howard, Jeremy, and Sylvain Gugger. 2020. "Fastai: A Layered API for Deep Learning." *Information* 11 (2): 108. https://doi.org/10.3390/info11020108.

Hsiao, Yu-Shun, Zishen Wan, Tianyu Jia, Radhika Ghosal, Abdulrahman Mahmoud, Arijit Raychowdhury, David Brooks, Gu-Yeon Wei, and Vijay Janapa Reddi. 2023. "MAVFI: An End-to-End Fault Analysis Framework with Anomaly Detection and Recovery for Micro Aerial Vehicles." In *2023 Design, Automation &Amp; Test in Europe Conference &Amp; Exhibition (DATE)*, 1–6. IEEE; IEEE. https://doi.org/10.23919/date56975.2023.10137246.

Hsu, Liang-Ching, Ching-Yi Huang, Yen-Hsun Chuang, Ho-Wen Chen, Ya-Ting Chan, Heng Yi Teah, Tsan-Yao Chen, Chiung-Fen Chang, Yu-Ting Liu, and Yu-Min Tzou. 2016. "Accumulation of Heavy Metals and Trace Elements in Fluvial Sediments Received Effluents from Traditional and Semiconductor Industries." *Scientific Reports* 6 (1): 34250. https://doi.org/10.1038/srep34250.

Hu, Bowen, Zhiqiang Zhang, and Yun Fu. 2021. "Triple Wins: Boosting Accuracy, Robustness and Efficiency Together by Enabling Input-Adaptive Inference." *Advances in Neural Information Processing Systems* 34: 18537–50.

Hu, Edward J., Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. "LoRA: Low-Rank Adaptation of Large Language Models." *arXiv Preprint arXiv:2106.09685*, June. http://arxiv.org/abs/2106.09685v2.

Hu, Jie, Peng Lin, Huajun Zhang, Zining Lan, Wenxin Chen, Kailiang Xie, Siyun Chen, Hao Wang, and Sheng Chang. 2023. "A Dynamic Pruning Method on Multiple Sparse Structures in Deep Neural Networks." *IEEE Access* 11: 38448–57. https://doi.org/10.1109/access.2023.3267469.

Huang, Wei, Jie Chen, and Lei Zhang. 2023. "Adaptive Neural Networks for Real-Time Processing in Autonomous Systems." *IEEE Transactions on Intelligent Transportation Systems*.

Huang, Yanping et al. 2019. "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism." In *Advances in Neural Information Processing Systems (NeurIPS)*.

Hubara, Itay, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2018. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations." *Journal of Machine Learning Research (JMLR)* 18: 1–30.

Hutter, Frank, Lars Kotthoff, and Joaquin Vanschoren. 2019a. *Automated Machine Learning: Methods, Systems, Challenges*. *Automated Machine Learning*. Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5.

———. 2019b. *Automated Machine Learning: Methods, Systems, Challenges*. Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5.

Hutter, Michael, Jorn-Marc Schmidt, and Thomas Plos. 2009. "Contact-Based Fault Injections and Power Analysis on RFID Tags." In *2009 European Conference on Circuit Theory and Design*, 409–12. IEEE; IEEE. https://doi.org/10.1109/ecctd.2009.5275012.

Hwu, Wen-mei W. 2011. "Introduction." In *GPU Computing Gems Emerald Edition*, xix–xx. Elsevier. https://doi.org/10.1016/b978-0-12-384988-5.00064-4.

Iandola, Forrest N., Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. "SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size," February. http://arxiv.org/abs/1602.07360v4.

Inan, Hakan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, et al. 2023. "Llama Guard: LLM-Based Input-Output Safeguard for Human-AI Conversations," December. http://arxiv.org/abs/2312.06674v1.

Inc., Framework Computer. 2022. "Modular Laptops: A New Approach to Sustainable Computing."

Inc., Tesla. 2021. "Tesla AI Day: D1 Dojo Chip." *Tesla AI Day Presentation*.

Inmon, W. H. 2005. *Building the Data Warehouse*. John Wiley Sons.

Ioffe, Sergey, and Christian Szegedy. 2015a. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *International Conference on Machine Learning*, 448–56.

———. 2015b. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *International Conference on Machine Learning (ICML)*, February, 448–56. http://arxiv.org/abs/1502.03167v3.

Ippolito, Daphne, Florian Tramer, Milad Nasr, Chiyuan Zhang, Matthew Jagielski, Katherine Lee, Christopher Choquette Choo, and Nicholas Carlini. 2023. "Preventing Generation of Verbatim Memorization in Language Models Gives a False Sense of Privacy." In *Proceedings of the 16th International Natural Language Generation Conference*, 28–53. Association for Computational Linguistics. https://doi.org/10.18653/v1/2023.inlg-main.3.

Jacob, Benoit, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018a. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference." In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2704–13. IEEE. https://doi.org/10.1109/cvpr.2018.00286.

———. 2018c. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference." In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2704–13. IEEE. https://doi.org/10.1109/cvpr.2018.00286.

———. 2018b. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference." In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2704–13. IEEE. https://doi.org/10.1109/cvpr.2018.00286.

Jacobs, David, Bas Rokers, Archisman Rudra, and Zili Liu. 2002. "Fragment Completion in Humans and Machines." In *Advances in Neural Information Processing Systems 14*, 35:27–34. The MIT Press. https://doi.org/10.7551/mitpress/1120.003.0008.

Jaech, Aaron, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, et al. 2024. "OpenAI O1 System Card." *CoRR*. https://doi.org/10.48550/ARXIV.2412.16720.

Janapa Reddi, Vijay et al. 2022. "MLPerf Mobile V2. 0: An Industry-Standard Benchmark Suite for Mobile Machine Learning." In *Proceedings of Machine Learning and Systems*, 4:806–23.

Jevons, William Stanley. 1865. *The Coal Question: An Inquiry Concerning the Progress of the Nation, and the Probable Exhaustion of Our Coal Mines*. London: Macmillan; Co. https://www.econlib.org/library/YPDBooks/Jevons/jvnCQ.html.

Jha, A. R. 2014. *Rare Earth Materials: Properties and Applications*. CRC Press. https://doi.org/10.1201/b17045.

Jha, Saurabh, Subho Banerjee, Timothy Tsai, Siva K. S. Hari, Michael B. Sullivan, Zbigniew T. Kalbarczyk, Stephen W. Keckler, and Ravishankar K. Iyer. 2019. "ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection." In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 112–24. IEEE; IEEE. https://doi.org/10.1109/dsn.2019.00025.

Jia, Xianyan, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, et al. 2018. "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes." *arXiv Preprint arXiv:1807.11205*, July. http://arxiv.org/abs/1807.11205v1.

Jia, Xu, Bert De Brabandere, Tinne Tuytelaars, and Luc Van Gool. 2016. "Dynamic Filter Networks." *Advances in Neural Information Processing Systems* 29.

Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. "Caffe: Convolutional Architecture for Fast Feature Embedding." In *Proceedings of the 22nd ACM International Conference on Multimedia*, 675–78. ACM. https://doi.org/10.1145/2647868.2654889.

Jia, Zhihao, Matei Zaharia, and Alex Aiken. 2018. "Beyond Data and Model Parallelism for Deep Neural Networks." *arXiv Preprint arXiv:1807.05358*, July. http://arxiv.org/abs/1807.05358v1.

Jia, Ziheng, Nathan Tillman, Luis Vega, Po-An Ouyang, Matei Zaharia, and Joseph E. Gonzalez. 2019. "Optimizing DNN Computation with Relaxed Graph Substitutions." *Conference on Machine Learning and Systems (MLSys)*.

Jiao, Xiaoqi, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. "TinyBERT: Distilling BERT for Natural Language Understanding." In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.findings-emnlp.372.

Johnson, Rebecca. 2018. "The Right to Repair Movement and Its Implications for AI Hardware Longevity." *Technology and Society Review* 20 (4): 87–102.

Johnson-Roberson, Matthew, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. 2017. "Driving in the Matrix: Can Virtual Worlds Replace Human-Generated Annotations for Real World Tasks?" In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 746–53. Singapore, Singapore: IEEE. https://doi.org/10.1109/icra.2017.7989092.

Jones, Gareth A. 2018. "Joining Dessins Together." *arXiv Preprint arXiv:1810.03960*, October. http://arxiv.org/abs/1810.03960v1.

Jones, Nicholas P., Mark Johnson, and Claire Montgomery. 2021. "The Environmental Impact of Data Centers: Challenges and Sustainable Solutions." *Energy Reports* 7: 4381–92.

Jordan, T. L. 1982. "A Guide to Parallel Computation and Some Cray-1 Experiences." In *Parallel Computations*, 1–50. Elsevier. https://doi.org/10.1016/b978-0-12-592101-5.50006-3.

Joulin, Armand, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2017. "Bag of Tricks for Efficient Text Classification." In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, 18:1–42. Association for Computational Linguistics. https://doi.org/10.18653/v1/e17-2068.

Jouppi, Norman P. et al. 2017. "In-Datacenter Performance Analysis of a Tensor Processing Unit." *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.

Jouppi, Norman P., Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, et al. 2021b. "Ten Lessons from Three Generations Shaped Google's TPUv4i : Industrial Product." In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 64:1–14. 5. IEEE. https://doi.org/10.1109/isca52012.2021.00010.

———, et al. 2021a. "Ten Lessons from Three Generations Shaped Google's TPUv4i : Industrial Product." In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 1–14. IEEE. https://doi.org/10.1109/isca52012.2021.00010.

Jouppi, Norman P., Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. "A Domain-Specific Supercomputer for Training Deep Neural Networks." *Communications of the ACM* 63 (7): 67–78. https://doi.org/10.1145/3360307.

Jouppi, Norman P., Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, et al. 2017a. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 1–12. ACM. https://doi.org/10.1145/3079856.3080246.

———, et al. 2017c. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 1–12. ACM. https://doi.org/10.1145/3079856.3080246.

———, et al. 2017b. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 1–12. ACM. https://doi.org/10.1145/3079856.3080246.

Joye, Marc, and Michael Tunstall. 2012. *Fault Analysis in Cryptography*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-29656-7.

Kannan, Harish, Pradeep Dubey, and Mark Horowitz. 2023. "Chiplet-Based Architectures: The Future of AI Accelerators." *IEEE Micro* 43 (1): 46–55. https://doi.org/10.1109/MM.2022.1234567.

Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. "Scaling Laws for Neural Language Models." *ArXiv Preprint* abs/2001.08361 (January). http://arxiv.org/abs/2001.08361v1.

Kawazoe Aguilera, Marcos, Wei Chen, and Sam Toueg. 1997. "Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication." In *Distributed Algorithms*, 126–40. Springer; Springer Berlin Heidelberg. https://doi.org/10.1007/bfb0030680.

Kiela, Douwe, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, et al. 2021. "Dynabench: Rethinking Benchmarking in NLP." In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 9:418–34. Online: Association for Computational Linguistics. https://doi.org/10.18653/v1/2021.naacl-main.324.

Kim, Jungrae, Michael Sullivan, and Mattan Erez. 2015. "Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory." In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 101–12. IEEE; IEEE. https://doi.org/10.1109/hpca.2015.7056025.

Kim, Sunju, Chungsik Yoon, Seunghon Ham, Jihoon Park, Ohun Kwon, Donguk Park, Sangjun Choi, Seungwon Kim, Kwonchul Ha, and Won Kim. 2018. "Chemical Use in the Semiconductor Manufacturing Industry." *International Journal of Occupational and Environmental Health* 24 (3-4): 109–18. https://doi.org/10.1080/10773525.2018.1519957.

Kingma, Diederik P., and Jimmy Ba. 2014. "Adam: A Method for Stochastic Optimization." *ICLR*, December. http://arxiv.org/abs/1412.6980v9.

Kirkpatrick, James, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, et al. 2017. "Overcoming Catastrophic Forgetting in Neural Networks." *Proceedings of the National Academy of Sciences* 114 (13): 3521–26. https://doi.org/10.1073/pnas.1611835114.

Kleppmann, Martin. 2016. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media. http://shop.oreilly.com/product/0636920032175.do.

Ko, Yohan. 2021. "Characterizing System-Level Masking Effects Against Soft Errors." *Electronics* 10 (18): 2286. https://doi.org/10.3390/electronics10182286.

Kocher, Paul, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, et al. 2019b. "Spectre Attacks: Exploiting Speculative Execution." In *2019 IEEE Symposium on Security and Privacy (SP)*, 1–19. IEEE. https://doi.org/10.1109/sp.2019.00002.

———, et al. 2019a. "Spectre Attacks: Exploiting Speculative Execution." In *2019 IEEE Symposium on Security and Privacy (SP)*, 1–19. IEEE. https://doi.org/10.1109/sp.2019.00002.

Kocher, Paul, Joshua Jaffe, and Benjamin Jun. 1999. "Differential Power Analysis." In *Advances in Cryptology — CRYPTO' 99*, 388–97. Springer; Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-48405-1/_25.

Kocher, Paul, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. "Introduction to Differential Power Analysis." *Journal of Cryptographic Engineering* 1 (1): 5–27. https://doi.org/10.1007/s13389-011-0006-y.

Koh, Pang Wei, Thao Nguyen, Yew Siang Tang, Stephen Mussmann, Emma Pierson, Been Kim, and Percy Liang. 2020. "Concept Bottleneck Models." In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020,* *13-18 July 2020, Virtual Event*, 119:5338–48. Proceedings of Machine Learning Research. PMLR. http://proceedings.mlr.press/v119/koh20a.html.

Koh, Pang Wei, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, et al. 2021. "WILDS: A Benchmark of in-the-Wild Distribution Shifts." In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, edited by Marina Meila and Tong Zhang, 139:5637–64. Proceedings of Machine Learning Research. PMLR. http://proceedings.mlr.press/v139/koh21a.html.

Koizumi, Yuma, Shoichiro Saito, Hisashi Uematsu, Noboru Harada, and Keisuke Imoto. 2019. "ToyADMOS: A Dataset of Miniature-Machine Operating Sounds for Anomalous Sound Detection." In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, 313–17. IEEE; IEEE. https://doi.org/10.1109/waspaa.2019.8937164.

Konecný, Jakub, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. "Federated Optimization: Distributed Machine Learning for on-Device Intelligence." *CoRR*. http://arxiv.org/abs/1610.02527.

Kreuzberger, Dominik, Florian Kerschbaum, and Thomas Kuhn. 2022. "Machine Learning Operations (MLOps): Overview, Definition, and Architecture." *ACM Computing Surveys (CSUR)* 55 (5): 1–32. https://doi.org/10.1145/3533378.

Krishnamoorthi, Raghuraman. 2018. "Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper." *arXiv Preprint arXiv:1806.08342*, June. http://arxiv.org/abs/1806.08342v1.

Krishnan, Rayan, Pranav Rajpurkar, and Eric J. Topol. 2022. "Self-Supervised Learning in Medicine and Healthcare." *Nature Biomedical Engineering* 6 (12): 1346–52. https://doi.org/10.1038/s41551-022-00914-1.

Krizhevsky, Alex. 2009. "Learning Multiple Layers of Features from Tiny Images."

Krizhevsky, Alex, Geoffrey Hinton, et al. 2009. "Learning Multiple Layers of Features from Tiny Images."

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2017b. "ImageNet Classification with Deep Convolutional Neural Networks." *Communications of the ACM* 60 (6): 84–90. https://doi.org/10.1145/3065386.

———. 2017c. "ImageNet Classification with Deep Convolutional Neural Networks." Edited by F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger. *Communications of the ACM* 60 (6): 84–90. https://doi.org/10.1145/3065386.

———. 2017a. "ImageNet Classification with Deep Convolutional Neural Networks." *Communications of the ACM* 60 (6): 84–90. https://doi.org/10.1145/3065386.

Kuchaiev, Oleksii, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Carl Case, and Paulius Micikevicius. 2018. "OpenSeq2Seq: Extensible Toolkit for Distributed and Mixed Precision Training of Sequence-to-Sequence Models." In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, 41–46. Association for Computational Linguistics. https://doi.org/10.18653/v1/w18-2507.

Kuhn, Max, and Kjell Johnson. 2013. *Applied Predictive Modeling*. Springer New York. https://doi.org/10.1007/978-1-4614-6849-3.

Kung. 1982. "Why Systolic Architectures?" *Computer* 15 (1): 37–46. https://doi.org/10.1109/mc.1982.1653825.

Kung, Hsiang Tsung, and Charles E Leiserson. 1979. "Systolic Arrays (for VLSI)." In *Sparse Matrix Proceedings 1978*, 1:256–82. Society for industrial; applied mathematics Philadelphia, PA, USA.

Labarge, Isaac E. n.d. "Neural Network Pruning for ECG Arrhythmia Classification." *Proceedings of Machine Learning and Systems (MLSys)*. PhD thesis, California Polytechnic State University. https://doi.org/10.15368/theses.2020.76.

Lai, Liangzhen, Naveen Suda, and Vikas Chandra. 2018. "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-m CPUs." *ArXiv Preprint* abs/1801.06601 (January). http://arxiv.org/abs/1801.06601v1.

Lai, Pete Warden Daniel Situnayake. 2020. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media.

Lam, Monica D., Edward E. Rothberg, and Michael E. Wolf. 1991. "The Cache Performance and Optimizations of Blocked Algorithms." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS-IV*, 63–74. ACM Press. https://doi.org/10.1145/106972.106981.

Lange, Klaus-Dieter. 2009. "Identifying Shades of Green: The SPECpower Benchmarks." *Computer* 42 (3): 95–97. https://doi.org/10.1109/mc.2009.84.

Lattner, Chris, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. "MLIR: A Compiler Infrastructure for the End of Moore's Law." *arXiv Preprint arXiv:2002.11054*, February. http://arxiv.org/abs/2002.11054v2.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015a. "Deep Learning." *Nature* 521 (7553): 436–44. https://doi.org/10.1038/nature14539.

———. 2015b. "Deep Learning." *Nature* 521 (7553): 436–44. https://doi.org/10.1038/nature14539.

LeCun, Yann, Leon Bottou, Genevieve B. Orr, and Klaus -Robert Müller. 1998. "Efficient BackProp." In *Neural Networks: Tricks of the Trade*, 1524:9–50. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-49430-8/_2.

LeCun, Yann, John S. Denker, and Sara A. Solla. 1989. "Optimal Brain Damage." In *Advances in Neural Information Processing Systems*, 2:598–605. Morgan-Kaufmann. http://papers.nips.cc/paper/250-optimal-brain-damage.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. "Backpropagation Applied to Handwritten Zip Code Recognition." *Neural Computation* 1 (4): 541–51. https://doi.org/10.1162/neco.1989.1.4.541.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–2324. https://doi.org/10.1109/5.726791.

Lee, Minwoong, Namho Lee, Huijeong Gwon, Jongyeol Kim, Younggwan Hwang, and Seongik Cho. 2022. "Design of Radiation-Tolerant High-Speed Signal Processing Circuit for Detecting Prompt Gamma Rays by Nuclear

Explosion." *Electronics* 11 (18): 2970. https://doi.org/10.3390/electronics11182970.

Lepikhin, Dmitry et al. 2020. "GShard: Scaling Giant Models with Conditional Computation." In *Proceedings of the International Conference on Learning Representations*.

LeRoy Poff, N, MM Brinson, and JW Day. 2002. "Aquatic Ecosystems & Global Climate Change." *Pew Center on Global Climate Change*.

Levy, Orin, Alon Cohen, Asaf Cassel, and Yishay Mansour. 2023. "Efficient Rate Optimal Regret for Adversarial Contextual MDPs Using Online Function Approximation." *arXiv Preprint arXiv:2303.01464*, March. http://arxiv.org/abs/2303.01464v2.

Li, Fengfu, Bin Liu, Xiaoxing Wang, Bo Zhang, and Junchi Yan. 2016. "Ternary Weight Networks." *arXiv Preprint*, May. http://arxiv.org/abs/1605.04711v3.

Li, Guanpeng, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. 2017. "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12. ACM. https://doi.org/10.1145/3126908.3126964.

Li, Lisha, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization." *J. Mach. Learn. Res.* 18: 185:1–52. https://jmlr.org/papers/v18/16-558.html.

Li, Qinbin, Zeyi Wen, Zhaomin Wu, Sixu Hu, Naibo Wang, Yuan Li, Xu Liu, and Bingsheng He. 2023. "A Survey on Federated Learning Systems: Vision, Hype and Reality for Data Privacy and Protection." *IEEE Transactions on Knowledge and Data Engineering* 35 (4): 3347–66. https://doi.org/10.1109/tkde.2021.3124599.

Li, Tian, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. "Federated Learning: Challenges, Methods, and Future Directions." *IEEE Signal Processing Magazine* 37 (3): 50–60. https://doi.org/10.1109/msp.2020.2975749.

Li, Zhuohan, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, et al. 2023. "{AlpaServe}: Statistical Multiplexing with Model Parallelism for Deep Learning Serving." In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 663–79.

Liang, Percy, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, et al. 2022. "Holistic Evaluation of Language Models." *arXiv Preprint arXiv:2211.09110*, November. http://arxiv.org/abs/2211.09110v2.

Lin, Ji, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. "MCUNet: Tiny Deep Learning on IoT Devices." In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, Virtual*, edited by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. https://proceedings.neurips.cc/paper/2020/hash/86c51678350f656dcc7f490a43946ee5-Abstract.html.

Lin, Jiong, Qing Gao, Yungui Gong, Yizhou Lu, Chao Zhang, and Fengge Zhang. 2020. "Primordial Black Holes and Secondary Gravitational Waves from k/g Inflation." *arXiv Preprint arXiv:2001.05909*, January. http://arxiv.org/abs/2001.05909v2.

Lin, Ji, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2023. "AWQ: Activation-Aware Weight Quantization for LLM Compression and Acceleration." *arXiv Preprint arXiv:2306.00978* abs/2306.00978 (June). http://arxiv.org/abs/2306.00978v5.

Lin, Ji, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, and Song Han. 2023. "Tiny Machine Learning: Progress and Futures [Feature]." *IEEE Circuits and Systems Magazine* 23 (3): 8–34. https://doi.org/10.1109/mcas.2023.3302182.

Lin, Tsung-Yi, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. "Microsoft COCO: Common Objects in Context." In *Computer Vision – ECCV 2014*, 740–55. Springer; Springer International Publishing. https://doi.org/10.1007/978-3-319-10602-1/_48.

Lindgren, Simon. 2023. *Handbook of Critical Studies of Artificial Intelligence*. Edward Elgar Publishing.

Lindholm, Andreas, Dave Zachariah, Petre Stoica, and Thomas B. Schon. 2019. "Data Consistency Approach to Model Validation." *IEEE Access* 7: 59788–96. https://doi.org/10.1109/access.2019.2915109.

Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. 2008. "NVIDIA Tesla: A Unified Graphics and Computing Architecture." *IEEE Micro* 28 (2): 39–55. https://doi.org/10.1109/mm.2008.31.

Liu, Chen, Guillaume Bellec, Bernhard Vogginger, David Kappel, Johannes Partzsch, Felix Neumärker, Sebastian Höppner, et al. 2018. "Memory-Efficient Deep Learning on a SpiNNaker 2 Prototype." *Frontiers in Neuroscience* 12 (November): 840. https://doi.org/10.3389/fnins.2018.00840.

Liu, Yanan, Xiaoxia Wei, Jinyu Xiao, Zhijie Liu, Yang Xu, and Yun Tian. 2020. "Energy Consumption and Emission Mitigation Prediction Based on Data Center Traffic and PUE for Global Data Centers." *Global Energy Interconnection* 3 (3): 272–82. https://doi.org/10.1016/j.gloei.2020.07.008.

Liu, Yingcheng, Guo Zhang, Christopher G. Tarolli, Rumen Hristov, Stella Jensen-Roberts, Emma M. Waddell, Taylor L. Myers, et al. 2022. "Monitoring Gait at Home with Radio Waves in Parkinson's Disease: A Marker of Severity, Progression, and Medication Response." *Science Translational Medicine* 14 (663): eadc9669. https://doi.org/10.1126/scitranslmed.adc9669.

Lopez-Paz, David, and Marc'Aurelio Ranzato. 2017. "Gradient Episodic Memory for Continual Learning." In *NIPS*, 30:6467–76. https://proceedings.neurips.cc/paper/2017/hash/f87522788a2be2d171666752f97ddebb-Abstract.html.

Lu, Yucheng, Shivani Agrawal, Suvinay Subramanian, Oleg Rybakov, Christopher De Sa, and Amir Yazdanbakhsh. 2023. "STEP: Learning n:m Structured Sparsity Masks from Scratch with Precondition," February. http://arxiv.org/abs/2302.01172v1.

Luna, William Fernando Martínez. 2018a. "CONSUMER PROTECTION AGAINST PLANNED OBSOLESCENCE. AN INTERNATIONAL PRIVATE LAW ANAL-

YSIS." In *Planned Obsolescence and the Rulle of Law*, 12:229–80. 3. Universidad del Externado de Colombia. https://doi.org/10.2307/j.ctv1ddcwvh.9.

———. 2018b. "CONSUMER PROTECTION AGAINST PLANNED OBSOLESCENCE. AN INTERNATIONAL PRIVATE LAW ANALYSIS." In *Planned Obsolescence and the Rulle of Law*, 15:229–80. 2. Universidad del Externado de Colombia. https://doi.org/10.2307/j.ctv1ddcwvh.9.

Lundberg, Scott M., and Su-In Lee. 2017. "A Unified Approach to Interpreting Model Predictions." In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, edited by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, 4765–74. https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html.

Lyons, Richard G. 2011. *Understanding Digital Signal Processing*. 3rd ed. Prentice Hall.

Ma, Dongning, Fred Lin, Alban Desmaison, Joel Coburn, Daniel Moore, Sriram Sankar, and Xun Jiao. 2024. "Dr. DNA: Combating Silent Data Corruptions in Deep Learning Using Distribution of Neuron Activations." In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 239–52. ACM. https://doi.org/10.1145/3620666.3651349.

Ma, Jeffrey, Alan Tu, Yiling Chen, and Vijay Janapa Reddi. 2024. "FedStaleWeight: Buffered Asynchronous Federated Learning with Fair Aggregation via Staleness Reweighting," June. http://arxiv.org/abs/2406.02877v1.

Maas, Martin, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2024. "Combining Machine Learning and Lifetime-Based Resource Management for Memory Allocation and Beyond." *Communications of the ACM* 67 (4): 87–96. https://doi.org/10.1145/3611018.

Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. "Towards Deep Learning Models Resistant to Adversarial Attacks." *arXiv Preprint arXiv:1706.06083*, June. http://arxiv.org/abs/1706.06083v4.

Mahmoud, Abdulrahman, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. 2020. "PyTorchFI: A Runtime Perturbation Tool for DNNs." In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-w)*, 25–31. IEEE; IEEE. https://doi.org/10.1109/dsn-w50199.2020.00014.

Mahmoud, Abdulrahman, Siva Kumar Sastry Hari, Christopher W. Fletcher, Sarita V. Adve, Charbel Sakr, Naresh Shanbhag, Pavlo Molchanov, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. 2021. "Optimizing Selective Protection for CNN Resilience." In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 127–38. IEEE. https://doi.org/10.1109/issre52982.2021.00025.

Mahmoud, Abdulrahman, Thierry Tambe, Tarek Aloui, David Brooks, and Gu-Yeon Wei. 2022. "GoldenEye: A Platform for Evaluating Emerging Numerical Data Formats in DNN Accelerators." In *2022 52nd Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks (DSN), 206–14. IEEE. https://doi.org/10.1109/dsn53405.2022.00031.

Martin, C. Dianne. 1993. "The Myth of the Awesome Thinking Machine." *Communications of the ACM* 36 (4): 120–33. https://doi.org/10.1145/255950.153587.

Marulli, Fiammetta, Stefano Marrone, and Laura Verde. 2022. "Sensitivity of Machine Learning Approaches to Fake and Untrusted Data in Healthcare Domain." *Journal of Sensor and Actuator Networks* 11 (2): 21. https://doi.org/10.3390/jsan11020021.

Masanet, Eric, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. 2020b. "Recalibrating Global Data Center Energy-Use Estimates." *Science* 367 (6481): 984–86. https://doi.org/10.1126/science.aba3758.

———. 2020a. "Recalibrating Global Data Center Energy-Use Estimates." *Science* 367 (6481): 984–86. https://doi.org/10.1126/science.aba3758.

Maslej, Nestor, Loredana Fattorini, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, et al. 2023. "Artificial Intelligence Index Report 2023." *ArXiv Preprint* abs/2310.03715 (October). http://arxiv.org/abs/2310.03715v1.

Maslej, Nestor, Loredana Fattorini, C. Raymond Perrault, Vanessa Parli, Anka Reuel, Erik Brynjolfsson, John Etchemendy, et al. 2024. "Artificial Intelligence Index Report 2024." *CoRR*. https://doi.org/10.48550/ARXIV.2405.19522.

Mattson, Peter, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, et al. 2020. "MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance." *IEEE Micro* 40 (2): 8–16. https://doi.org/10.1109/mm.2020.2974843.

Mazumder, Mark, Sharad Chitlangia, Colby Banbury, Yiping Kang, Juan Manuel Ciro, Keith Achorn, Daniel Galvez, et al. 2021. "Multilingual Spoken Words Corpus." In *Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

McAuliffe, Michael, Michaela Socolof, Sarah Mihuc, Michael Wagner, and Morgan Sonderegger. 2017. "Montreal Forced Aligner: Trainable Text-Speech Alignment Using Kaldi." In *Interspeech 2017*, 498–502. ISCA. https://doi.org/10.21437/interspeech.2017-1386.

McCarthy, John. 1981. "EPISTEMOLOGICAL PROBLEMS OF ARTIFICIAL INTELLIGENCE." In *Readings in Artificial Intelligence*, 459–65. Elsevier. https://doi.org/10.1016/b978-0-934613-03-3.50035-0.

McMahan, Brendan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017a. "Communication-Efficient Learning of Deep Networks from Decentralized Data." In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, edited by Aarti Singh and Xiaojin (Jerry) Zhu, 54:1273–82. Proceedings of Machine Learning Research. PMLR. http://proceedings.mlr.press/v54/mcmahan17a.html.

———. 2017b. "Communication-Efficient Learning of Deep Networks from Decentralized Data." In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 1273–82. PMLR. http://proceedings.mlr.press/v54/mcmahan17a.html.

McMahan, H Brendan, Eider Moore, Daniel Ramage, Seth Hampson, et al. 2017. "Communication-Efficient Learning of Deep Networks from Decentralized Data." In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 1273–82.

Mellempudi, Naveen, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. 2019. "Mixed Precision Training with 8-Bit Floating Point." *arXiv Preprint arXiv:1905.12334*, May. http://arxiv.org/abs/1905.12334v1.

Merity, Stephen, Caiming Xiong, James Bradbury, and Richard Socher. 2016. "Pointer Sentinel Mixture Models." *arXiv Preprint arXiv:1609.07843*, September. http://arxiv.org/abs/1609.07843v1.

Micikevicius, Paulius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, et al. 2017b. "Mixed Precision Training." *arXiv Preprint arXiv:1710.03740*, October. http://arxiv.org/abs/1710.03740v3.

———, et al. 2017a. "Mixed Precision Training." *arXiv Preprint arXiv:1710.03740*, October. http://arxiv.org/abs/1710.03740v3.

Micikevicius, Paulius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, et al. 2022. "FP8 Formats for Deep Learning." *arXiv Preprint arXiv:2209.05433*, September. http://arxiv.org/abs/2209.05433v2.

Miller, Charlie. 2019. "Lessons Learned from Hacking a Car." *IEEE Design &Amp; Test* 36 (6): 7–9. https://doi.org/10.1109/mdat.2018.2863106.

Miller, Charlie, and Chris Valasek. 2015. "The Antivirus Hacker's Handbook." *Black Hat USA*. Wiley. https://doi.org/10.1002/9781119183525.ch15.

Mills, Andrew, and Stephen Le Hunte. 1997. "An Overview of Semiconductor Photocatalysis." *Journal of Photochemistry and Photobiology A: Chemistry* 108 (1): 1–35. https://doi.org/10.1016/s1010-6030(97)00118-4.

Mirhoseini, Azalia et al. 2017. "Device Placement Optimization with Reinforcement Learning." *International Conference on Machine Learning (ICML)*.

Mohanram, K., and N. A. Touba. n.d. "Partial Error Masking to Reduce Soft Error Failure Rate in Logic Circuits." In *Proceedings. 16th IEEE Symposium on Computer Arithmetic*, 433–40. IEEE; IEEE Comput. Soc. https://doi.org/10.1109/dftvs.2003.1250141.

Moore, Gordon. 2021. "Cramming More Components onto Integrated Circuits (1965)." In *Ideas That Created the Future*, 261–66. The MIT Press. https://doi.org/10.7551/mitpress/12274.003.0027.

Mukherjee, S. S., J. Emer, and S. K. Reinhardt. n.d. "The Soft Error Problem: An Architectural Perspective." In *11th International Symposium on High-Performance Computer Architecture*, 243–47. IEEE; IEEE. https://doi.org/10.1109/hpca.2005.37.

Nagel, Markus, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. 2021b. "A White Paper on Neural Network Quantization." *arXiv Preprint arXiv:2106.08295*, June. http://arxiv.org/abs/2106.08295v1.

———. 2021a. "A White Paper on Neural Network Quantization." *arXiv Preprint arXiv:2106.08295*, June. http://arxiv.org/abs/2106.08295v1.

Narang, Sharan, Hyung Won Chung, Yi Tay, Liam Fedus, Thibault Fevry, Michael Matena, Karishma Malkan, et al. 2021. "Do Transformer Modifications Transfer Across Implementations and Applications?" In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 21:1–67. 140. Association for Computational Linguistics. https://doi.org/10.18653/v1/2021.emnlp-main.465.

Narayanan, Arvind, and Vitaly Shmatikov. 2006. "How to Break Anonymity of the Netflix Prize Dataset." *CoRR*. http://arxiv.org/abs/cs/0610105.

Narayanan, Deepak, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, et al. 2021a. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM." *NeurIPS*, April. http://arxiv.org/abs/2104.04473v5.

Narayanan, Deepak, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, et al. 2021b. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–15. ACM. https://doi.org/10.1145/3458817.3476209.

Nayak, Prateeth, Takuya Higuchi, Anmol Gupta, Shivesh Ranjan, Stephen Shum, Siddharth Sigtia, Erik Marchi, et al. 2022. "Improving Voice Trigger Detection with Metric Learning." *arXiv Preprint arXiv:2204.02455*, April. http://arxiv.org/abs/2204.02455v2.

Ng, Davy Tsz Kit, Jac Ka Lok Leung, Kai Wah Samuel Chu, and Maggie Shen Qiao. 2021. "<Scp>AI</Scp> Literacy: Definition, Teaching, Evaluation and Ethical Issues." *Proceedings of the Association for Information Science and Technology* 58 (1): 504–9. https://doi.org/10.1002/pra2.487.

Ngo, Richard, Lawrence Chan, and Sören Mindermann. 2022. "The Alignment Problem from a Deep Learning Perspective." *ArXiv Preprint* abs/2209.00626 (August). http://arxiv.org/abs/2209.00626v8.

Nguyen, John, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Michael Rabbat, Mani Malek, and Dzmitry Huba. 2021. "Federated Learning with Buffered Asynchronous Aggregation," June. http://arxiv.org/abs/2106.06639v4.

Nishigaki, Shinsuke. 2024. "Eigenphase Distributions of Unimodular Circular Ensembles." *arXiv Preprint arXiv:2401.09045* 36 (January). http://arxiv.org/abs/2401.09045v2.

Norrie, Thomas, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. "The Design Process for Google's Training Chips: TPUv2 and TPUv3." *IEEE Micro* 41 (2): 56–63. https://doi.org/10.1109/mm.2021.3058217.

Northcutt, Curtis G, Anish Athalye, and Jonas Mueller. 2021. "Pervasive Label Errors in Test Sets Destabilize Machine Learning Benchmarks." *arXiv*. https://doi.org/https://doi.org/10.48550/arXiv.2103.14749 arXiv-issued DOI via DataCite.

NVIDIA. 2021. "TensorRT: High-Performance Deep Learning Inference Library." *NVIDIA Developer Blog*. https://developer.nvidia.com/tensorrt.

Oakden-Rayner, Luke, Jared Dunnmon, Gustavo Carneiro, and Christopher Re. 2020. "Hidden Stratification Causes Clinically Meaningful Failures in Machine Learning for Medical Imaging." In *Proceedings of the ACM Conference* *on Health, Inference, and Learning*, 151–59. ACM. https://doi.org/10.1145/3368555.3384468.

Obermeyer, Ziad, Brian Powers, Christine Vogeli, and Sendhil Mullainathan. 2019. "Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations." *Science* 366 (6464): 447–53. https://doi.org/10.1126/science.aax2342.

Oecd. 2023. "A Blueprint for Building National Compute Capacity for Artificial Intelligence." 350. Organisation for Economic Co-Operation; Development (OECD). https://doi.org/10.1787/876367e3-en.

OECD.AI. 2021. "Measuring the Geographic Distribution of AI Computing Capacity." <https://oecd.ai/en/policy-circle/computing-capacity>.

Olah, Chris, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. 2020. "Zoom in: An Introduction to Circuits." *Distill* 5 (3): e00024–001. https://doi.org/10.23915/distill.00024.001.

Oprea, Alina, Anoop Singhal, and Apostol Vassilev. 2022. "Poisoning Attacks Against Machine Learning: Can Machine Learning Be Trustworthy?" *Computer* 55 (11): 94–99. https://doi.org/10.1109/mc.2022.3190787.

Orekondy, Tribhuvanesh, Bernt Schiele, and Mario Fritz. 2019. "Knockoff Nets: Stealing Functionality of Black-Box Models." In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 4949–58. IEEE. https://doi.org/10.1109/cvpr.2019.00509.

Owens, J. D., M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. 2008. "GPU Computing." *Proceedings of the IEEE* 96 (5): 879–99. https://doi.org/10.1109/jproc.2008.917757.

Palmer, John F. 1980. "The INTEL® 8087 Numeric Data Processor." In *Proceedings of the May 19-22, 1980, National Computer Conference on - AFIPS '80*, 887. ACM Press. https://doi.org/10.1145/1500518.1500674.

Panda, Priyadarshini, Indranil Chakraborty, and Kaushik Roy. 2019. "Discretization Based Solutions for Secure Machine Learning Against Adversarial Attacks." *IEEE Access* 7: 70157–68. https://doi.org/10.1109/access.2019.2919463.

Papadimitriou, George, and Dimitris Gizopoulos. 2021. "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers." In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 902–15. IEEE; IEEE. https://doi.org/10.1109/isca52012.2021.00075.

Papernot, Nicolas, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks." In *2016 IEEE Symposium on Security and Privacy (SP)*, 582–97. IEEE; IEEE. https://doi.org/10.1109/sp.2016.41.

Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2001. "BLEU: A Method for Automatic Evaluation of Machine Translation." In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*, 311. Association for Computational Linguistics. https://doi.org/10.3115/1073083.1073135.

Park, Daniel S., William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D. Cubuk, and Quoc V. Le. 2019. "SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition." *arXiv Preprint arXiv:1904.08779*, April. http://arxiv.org/abs/1904.08779v3.

Parrish, Alicia, Hannah Rose Kirk, Jessica Quaye, Charvi Rastogi, Max Bartolo, Oana Inel, Juan Ciro, et al. 2023. "Adversarial Nibbler: A Data-Centric Challenge for Improving the Safety of Text-to-Image Models." *ArXiv Preprint* abs/2305.14384 (May). http://arxiv.org/abs/2305.14384v1.

Paszke, Adam, Sam Gross, Francisco Massa, and et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." *Advances in Neural Information Processing Systems (NeurIPS)* 32: 8026–37.

Patel, Paresh D., Absar Lakdawala, Sajan Chourasia, and Rajesh N. Patel. 2016. "Bio Fuels for Compression Ignition Engine: A Review on Engine Performance, Emission and Life Cycle Analysis." *Renewable and Sustainable Energy Reviews* 65 (November): 24–43. https://doi.org/10.1016/j.rser.2016.06.010.

Patterson, David A., and John L. Hennessy. 2021a. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann.

———. 2021b. *Computer Organization and Design RISC-v Edition: The Hardware Software Interface*. 2nd ed. San Francisco, CA: Morgan Kaufmann.

———. 2021c. *Computer Organization and Design: The Hardware/Software Interface*. 5th ed. Morgan Kaufmann.

Patterson, David, Joseph Gonzalez, Urs Holzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. 2022. "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink." *Computer* 55 (7): 18–28. https://doi.org/10.1109/mc.2022.3148714.

Patterson, David, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021a. "Carbon Emissions and Large Neural Network Training." *arXiv Preprint arXiv:2104.10350*, April. http://arxiv.org/abs/2104.10350v3.

———. 2021b. "Carbon Emissions and Large Neural Network Training." *arXiv Preprint arXiv:2104.10350*, April. http://arxiv.org/abs/2104.10350v3.

Patterson, David, Joseph Gonzalez, Quoc Le, Maud Texier, and Jeff Dean. 2022. "Carbon-Aware Computing for Sustainable AI." *Communications of the ACM* 65 (11): 50–58.

Penedo, Guilherme, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. 2024. "The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale." *arXiv Preprint arXiv:2406.17557*, June. http://arxiv.org/abs/2406.17557v2.

Peters, Dorian, Rafael A. Calvo, and Richard M. Ryan. 2018. "Designing for Motivation, Engagement and Wellbeing in Digital Experience." *Frontiers in Psychology* 9 (May): 797. https://doi.org/10.3389/fpsyg.2018.00797.

Phillips, P. Jonathon, Carina A. Hahn, Peter C. Fontana, David A. Broniatowski, and Mark A. Przybocki. 2020. "Four Principles of Explainable Artificial Intelligence." *Gaithersburg, Maryland*. National Institute of Standards; Technology (NIST). https://doi.org/10.6028/nist.ir.8312-draft.

Pineau, Joelle, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché-Buc, Emily Fox, and Hugo Larochelle. 2021. "Improving Reproducibility in Machine Learning Research (a Report from the Neurips 2019 Reproducibility Program)." *Journal of Machine Learning Research* 22 (164): 1–20.

Plank, James S. 1997. "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems." *Software: Practice and Experience* 27 (9): 995–1012. https://doi.org/10.1002/(sici)1097-024x(199709)27:9%3C995::aid-spe111%3E3.0.co;2-6.

Pont, Michael J, and Royan HL Ong. 2002. "Using Watchdog Timers to Improve the Reliability of Single-Processor Embedded Systems: Seven New Patterns and a Case Study." In *Proceedings of the First Nordic Conference on Pattern Languages of Programs*, 159–200. Citeseer.

Prakash, Shvetank, Tim Callahan, Joseph Bushagour, Colby Banbury, Alan V. Green, Pete Warden, Tim Ansell, and Vijay Janapa Reddi. 2023. "CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs." In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, abs/2201.01863:157–67. IEEE. https://doi.org/10.1109/ispass57527.2023.00024.

Psoma, Sotiria D., and Chryso Kanthou. 2023. "Wearable Insulin Biosensors for Diabetes Management: Advances and Challenges." *Biosensors* 13 (7): 719. https://doi.org/10.3390/bios13070719.

Puckett, Jim. 2016. *E-Waste and the Global Environment: The Hidden Cost of Discarded Electronics*. MIT Press.

Pushkarna, Mahima, Andrew Zaldivar, and Oddur Kjartansson. 2022. "Data Cards: Purposeful and Transparent Dataset Documentation for Responsible AI." In *2022 ACM Conference on Fairness, Accountability, and Transparency*, 1776–826. ACM. https://doi.org/10.1145/3531146.3533231.

Putnam, Andrew, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, et al. 2014. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services." *ACM SIGARCH Computer Architecture News* 42 (3): 13–24. https://doi.org/10.1145/2678373.2665678.

Qi, Chen, Shibo Shen, Rongpeng Li, Zhifeng Zhao, Qing Liu, Jing Liang, and Honggang Zhang. 2021. "An Efficient Pruning Scheme of Deep Neural Networks for Internet of Things Applications." *EURASIP Journal on Advances in Signal Processing* 2021 (1): 31. https://doi.org/10.1186/s13634-021-00744-4.

Qi, Xuan, Burak Kantarci, and Chen Liu. 2017. "GPU-Based Acceleration of SDN Controllers." In *Network as a Service for Next Generation Internet*, 339–56. Institution of Engineering; Technology. https://doi.org/10.1049/pbte073e/_ch14.

Quaye, Jessica, Alicia Parrish, Oana Inel, Charvi Rastogi, Hannah Rose Kirk, Minsuk Kahng, Erin Van Liemt, et al. 2024. "Adversarial Nibbler: An Open Red-Teaming Method for Identifying Diverse Harms in Text-to-Image Generation." In *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, 388–406. ACM. https://doi.org/10.1145/3630106.3658913.

Quiñonero-Candela, Joaquin, Masashi Sugiyama, Anton Schwaighofer, and Neil D. Lawrence. 2008. "Dataset Shift in Machine Learning." *The MIT Press*. The MIT Press. https://doi.org/10.7551/mitpress/7921.003.0002.

R. V., Rashmi, and Karthikeyan A. 2018. "Secure Boot of Embedded Applications - a Review." In *2018 Second International Conference on Electron-*

ics, Communication and Aerospace Technology (ICECA), 291–98. IEEE. https://doi.org/10.1109/iceca.2018.8474730.

Radosavovic, Ilija, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollar. 2020. "Designing Network Design Spaces." In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 10428–36. IEEE. https://doi.org/10.1109/cvpr42600.2020.01044.

Rainio, Oona, Jarmo Teuho, and Riku Klén. 2024. "Evaluation Metrics and Statistical Tests for Machine Learning." *Scientific Reports* 14 (1): 6086.

Rajbhandari, Samyam, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. "ZeRO: Memory Optimization Towards Training Trillion Parameter Models." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.5555/3433701.3433721.

Rajpurkar, Pranav, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. "SQuAD: 100,000+ Questions for Machine Comprehension of Text." *arXiv Preprint arXiv:1606.05250*, June, 2383–92. https://doi.org/10.18653/v1/d16-1264.

Ramcharan, Amanda, Kelsee Baranowski, Peter McCloskey, Babuali Ahmed, James Legg, and David P. Hughes. 2017. "Deep Learning for Image-Based Cassava Disease Detection." *Frontiers in Plant Science* 8 (October): 1852. https://doi.org/10.3389/fpls.2017.01852.

Ramesh, Aditya, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. "Zero-Shot Text-to-Image Generation." In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, edited by Marina Meila and Tong Zhang, 139:8821–31. Proceedings of Machine Learning Research. PMLR. http://proceedings.mlr.press/v139/ramesh21a.html.

Ranganathan, Parthasarathy, and Urs Hölzle. 2024. "Twenty Five Years of Warehouse-Scale Computing." *IEEE Micro* 44 (5): 11–22. https://doi.org/10.1109/mm.2024.3409469.

Rashid, Layali, Karthik Pattabiraman, and Sathish Gopalakrishnan. 2012. "Intermittent Hardware Errors Recovery: Modeling and Evaluation." In *2012 Ninth International Conference on Quantitative Evaluation of Systems*, 220–29. IEEE; IEEE. https://doi.org/10.1109/qest.2012.37.

———. 2015. "Characterizing the Impact of Intermittent Hardware Faults on Programs." *IEEE Transactions on Reliability* 64 (1): 297–310. https://doi.org/10.1109/tr.2014.2363152.

Rastegari, Mohammad, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks." In *Computer Vision – ECCV 2016*, 525–42. Springer International Publishing. https://doi.org/10.1007/978-3-319-46493-0/_32.

Ratner, Alex, Braden Hancock, Jared Dunnmon, Roger Goldman, and Christopher Ré. 2018. "Snorkel MeTaL: Weak Supervision for Multi-Task Learning." In *Proceedings of the Second Workshop on Data Management for End-to-End Machine Learning*. ACM. https://doi.org/10.1145/3209889.3209898.

Reagen, Brandon, Robert Adolf, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2017. *Deep Learning for Computer Architects*. Springer International Publishing. https://doi.org/10.1007/978-3-031-01756-8.

Reagen, Brandon, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. 2018. "Ares: A Framework for Quantifying the Resilience of Deep Neural Networks." In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 1–6. IEEE. https://doi.org/10.1109/dac.2018.8465834.

Real, Esteban, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019b. "Regularized Evolution for Image Classifier Architecture Search." *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (01): 4780–89. https://doi.org/10.1609/aaai.v33i01.33014780.

———. 2019a. "Regularized Evolution for Image Classifier Architecture Search." *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (01): 4780–89. https://doi.org/10.1609/aaai.v33i01.33014780.

Rebuffi, Sylvestre-Alvise, Hakan Bilen, and Andrea Vedaldi. 2017. "Learning Multiple Visual Domains with Residual Adapters." In *Advances in Neural Information Processing Systems*. Vol. 30.

Reddi, Vijay Janapa, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, et al. 2019. "MLPerf Inference Benchmark." *arXiv Preprint arXiv:1911.02549*, November, 446–59. https://doi.org/10.1109/isca45697.2020.00045.

Reddi, Vijay Janapa, and Meeta Sharma Gupta. 2013. *Resilient Architecture Design for Voltage Variation*. Springer International Publishing. https://doi.org/10.1007/978-3-031-01739-1.

Reis, G. A., J. Chang, N. Vachharajani, R. Rangan, and D. I. August. n.d. "SWIFT: Software Implemented Fault Tolerance." In *International Symposium on Code Generation and Optimization*, 243–54. IEEE; IEEE. https://doi.org/10.1109/cgo.2005.34.

Research, Microsoft. 2021. *DeepSpeed: Extreme-Scale Model Training for Everyone*.

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. 2016. "" Why Should i Trust You?" Explaining the Predictions of Any Classifier." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–44.

Richter, Joel D., and Xinyu Zhao. 2021. "The Molecular Biology of FMRP: New Insights into Fragile x Syndrome." *Nature Reviews Neuroscience* 22 (4): 209–22. https://doi.org/10.1038/s41583-021-00432-0.

Robertson, J., and M. Riley. 2018. "The Big Hack: How China Used a Tiny Chip to Infiltrate u.s. Companies - Bloomberg."https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies .

Rodio, Angelo, and Giovanni Neglia. 2024. "FedStale: Leveraging Stale Client Updates in Federated Learning," May. http://arxiv.org/abs/2405.04171v1.

Rolnick, David, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Greg Wayne. 2019. "Experience Replay for Continual Learning." In *Advances in Neural Information Processing Systems (NeurIPS)*.

Rombach, Robin, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Bjorn Ommer. 2022. "High-Resolution Image Synthesis with Latent Diffusion Models." In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 10674–85. IEEE. https://doi.org/10.1109/cvpr52688.2022.01042.

Romero, Francisco, Qian Li 0027, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. "INFaaS: Automated Model-Less Inference Serving." In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 397–411. https://www.usenix.org/conference/atc21/presentation/romero.

Rosenblatt, F. 1958. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." *Psychological Review* 65 (6): 386–408. https://doi.org/10.1037/h0042519.

Rudin, Cynthia. 2019. "Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead." *Nature Machine Intelligence* 1 (5): 206–15. https://doi.org/10.1038/s42256-019-0048-x.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088): 533–36. https://doi.org/10.1038/323533a0.

Russell, Mark. 2022. "Tech Industry Trends in Hardware Lock-in and Their Sustainability Implications." *Sustainable Computing Journal* 10 (1): 34–50.

Russell, Stuart. 2021. "Human-Compatible Artificial Intelligence." In *Human-Like Machine Intelligence*, 3–23. Oxford University Press. https://doi.org/10.1093/oso/9780198862536.003.0001.

Ryan, Richard M., and Edward L. Deci. 2000. "Self-Determination Theory and the Facilitation of Intrinsic Motivation, Social Development, and Well-Being." *American Psychologist* 55 (1): 68–78. https://doi.org/10.1037/0003-066x.55.1.68.

Sabour, Sara, Nicholas Frosst, and Geoffrey E Hinton. 2017. "Dynamic Routing Between Capsules." In *Advances in Neural Information Processing Systems*. Vol. 30.

Sambasivan, Nithya, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M Aroyo. 2021. "'Everyone Wants to Do the Model Work, Not the Data Work': Data Cascades in High-Stakes AI." In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 1–15. ACM. https://doi.org/10.1145/3411764.3445518.

Sangchoolie, Behrooz, Karthik Pattabiraman, and Johan Karlsson. 2017. "One Bit Is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors." In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 97–108. IEEE; IEEE. https://doi.org/10.1109/dsn.2017.30.

Sanh, Victor, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter." *arXiv Preprint arXiv:1910.01108*, October. http://arxiv.org/abs/1910.01108v4.

Savas, Esra, Reza Shokri, Lalith Singaravelu, Nithya Swamy, and Mitali Bafna. 2022. "ML-ExRay: Visibility and Explainability for Monitoring ML Model Behavior." In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*, 1352–69. IEEE.

Scardapane, Simone, Ye Wang, and Massimo Panella. 2020. "Why Should i Trust You? A Survey of Explainability of Machine Learning for Healthcare." *Pattern Recognition Letters* 140: 47–57.

Schäfer, Mike S. 2023. "The Notorious GPT: Science Communication in the Age of Artificial Intelligence." *Journal of Science Communication* 22 (02): Y02. https://doi.org/10.22323/2.22020402.

Schelter, Sebastian, Matthias Boehm, Johannes Kirschnick, Kostas Tzoumas, and Gunnar Ratsch. 2018. "Automating Large-Scale Machine Learning Model Management." In *Proceedings of the 2018 IEEE International Conference on Data Engineering (ICDE)*, 137–48. IEEE.

Schwartz, Roy, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. "Green AI." *Communications of the ACM* 63 (12): 54–63. https://doi.org/10.1145/3381831.

Sculley, D., G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. F. Crespo, and D. Dennison. 2015. "Hidden Technical Debt in Machine Learning Systems." In *Advances in Neural Information Processing Systems*. Vol. 28.

Seide, Frank, and Amit Agarwal. 2016. "CNTK: Microsoft's Open-Source Deep-Learning Toolkit." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2135–35. ACM. https://doi.org/10.1145/2939672.2945397.

Selvaraju, Ramprasaath R., Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization." In *2017 IEEE International Conference on Computer Vision (ICCV)*, 618–26. IEEE. https://doi.org/10.1109/iccv.2017.74.

Seong, Nak Hee, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A. Rivers, and Hsien-Hsin S. Lee. 2010. "SAFER: Stuck-at-Fault Error Recovery for Memories." In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 115–24. IEEE; IEEE. https://doi.org/10.1109/micro.2010.46.

Settles, Burr. 2012a. *Active Learning*. *Computer Sciences Technical Report*. University of Wisconsin–Madison; Springer International Publishing. https://doi.org/10.1007/978-3-031-01560-1.

———. 2012b. *Active Learning*. *University of Wisconsin-Madison Department of Computer Sciences*. Vol. 1648. Springer International Publishing. https://doi.org/10.1007/978-3-031-01560-1.

Sevilla, Jaime, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022a. "Compute Trends Across Three Eras of Machine Learning." In *2022 International Joint Conference on Neural Networks (IJCNN)*, 1–8. IEEE. https://doi.org/10.1109/ijcnn55064.2022.9891914.

———. 2022b. "Compute Trends Across Three Eras of Machine Learning." In *2022 International Joint Conference on Neural Networks (IJCNN)*, 1–8. IEEE. https://doi.org/10.1109/ijcnn55064.2022.9891914.

Shalev-Shwartz, Shai, Shaked Shammah, and Amnon Shashua. 2017. "On a Formal Model of Safe and Scalable Self-Driving Cars." *ArXiv Preprint* abs/1708.06374 (August). http://arxiv.org/abs/1708.06374v6.

Shallue, Christopher J., Jaehoon Lee, et al. 2019. "Measuring the Effects of Data Parallelism on Neural Network Training." *Journal of Machine Learning Research* 20: 1–49. http://jmlr.org/papers/v20/18-789.html.

Shan, Shawn, Wenxin Ding, Josephine Passananti, Stanley Wu, Haitao Zheng, and Ben Y. Zhao. 2023. "Nightshade: Prompt-Specific Poisoning Attacks on Text-to-Image Generative Models." *ArXiv Preprint* abs/2310.13828 (October). http://arxiv.org/abs/2310.13828v3.

Shang, J., G. Wang, and Y. Liu. 2018. "Accelerating Genomic Data Analysis with Domain-Specific Architectures." *IEEE Transactions on Computers* 67 (7): 965–78. https://doi.org/10.1109/TC.2018.2799212.

Sharma, Amit. 2020. "Industrial AI and Vendor Lock-in: The Hidden Costs of Proprietary Ecosystems." *AI and Industry Review* 8 (3): 55–70.

Shazeer, Noam, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, et al. 2018. "Mesh-TensorFlow: Deep Learning for Supercomputers." *arXiv Preprint arXiv:1811.02084*, November. http://arxiv.org/abs/1811.02084v1.

Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." *arXiv Preprint arXiv:1701.06538*, January. http://arxiv.org/abs/1701.06538v1.

Shazeer, Noam, Azalia Mirhoseini, Piotr Maziarz, et al. 2017. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." In *International Conference on Learning Representations*.

Sheaffer, Jeremy W., David P. Luebke, and Kevin Skadron. 2007. "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors." In *Graphics Hardware*, 2007:55–64. Citeseer. https://doi.org/10.2312/EGGH/EGGH07/055-064.

Shen, Sheng, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2019. "Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT." *Proceedings of the AAAI Conference on Artificial Intelligence* 34 (05): 8815–21. https://doi.org/10.1609/aaai.v34i05.6409.

Sheng, Victor S., and Jing Zhang. 2019. "Machine Learning with Crowdsourcing: A Brief Summary of the Past Research and Future Directions." *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (01): 9837–43. https://doi.org/10.1609/aaai.v33i01.33019837.

Shneiderman, Ben. 2020. "Bridging the Gap Between Ethics and Practice: Guidelines for Reliable, Safe, and Trustworthy Human-Centered AI Systems." *ACM Transactions on Interactive Intelligent Systems* 10 (4): 1–31. https://doi.org/10.1145/3419764.

———. 2022. *Human-Centered AI*. Oxford University Press.

Shoeybi, Mohammad, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019a. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." *arXiv Preprint arXiv:1909.08053*, September. http://arxiv.org/abs/1909.08053v4.

———. 2019b. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." *arXiv Preprint arXiv:1909.08053*, September. http://arxiv.org/abs/1909.08053v4.

Shokri, Reza, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. "Membership Inference Attacks Against Machine Learning Models." In

2017 IEEE Symposium on Security and Privacy (SP), 3–18. IEEE; IEEE. https://doi.org/10.1109/sp.2017.41.

Singh, Narendra, and Oladele A. Ogunseitan. 2022. "Disentangling the Worldwide Web of e-Waste and Climate Change Co-Benefits." *Circular Economy* 1 (2): 100011. https://doi.org/10.1016/j.cec.2022.100011.

Skorobogatov, Sergei. 2009. "Local Heating Attacks on Flash Memory Devices." In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 1–6. IEEE; IEEE. https://doi.org/10.1109/hst.2009.5225028.

Skorobogatov, Sergei P., and Ross J. Anderson. 2003. "Optical Fault Induction Attacks." In *Cryptographic Hardware and Embedded Systems - CHES 2002*, 2–12. Springer; Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-36400-5/_2.

Slade, Giles. 2007. *Made to Break: Technology and Obsolescence in America*. Harvard University Press. https://doi.org/10.4159/9780674043756.

Smith, Steven W. 1997. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing. https://www.dspguide.com/.

Sodani, Avinash. 2015. "Knights Landing (KNL): 2nd Generation Intel® Xeon Phi Processor." In *2015 IEEE Hot Chips 27 Symposium (HCS)*, 1–24. IEEE. https://doi.org/10.1109/hotchips.2015.7477467.

Sokolova, Marina, and Guy Lapalme. 2009. "A Systematic Analysis of Performance Measures for Classification Tasks." *Information Processing &Amp; Management* 45 (4): 427–37. https://doi.org/10.1016/j.ipm.2009.03.002.

Stahel, Walter R. 2016. "The Circular Economy." *Nature* 531 (7595): 435–38. https://doi.org/10.1038/531435a.

Statista. 2022. "Number of Internet of Things (IoT) Connected Devices Worldwide from 2019 to 2030." https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/.

Stephens, Nigel, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, et al. 2017. "The ARM Scalable Vector Extension." *IEEE Micro* 37 (2): 26–39. https://doi.org/10.1109/mm.2017.35.

Strassen, Volker. 1969. "Gaussian Elimination Is Not Optimal." *Numerische Mathematik* 13 (4): 354–56. https://doi.org/10.1007/bf02165411.

Strickland, Eliza. 2019. "IBM Watson, Heal Thyself: How IBM Overpromised and Underdelivered on AI Health Care." *IEEE Spectrum* 56 (4): 24–31. https://doi.org/10.1109/mspec.2019.8678513.

Strubell, Emma, Ananya Ganesh, and Andrew McCallum. 2019a. "Energy and Policy Considerations for Deep Learning in NLP." In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 3645–50. Association for Computational Linguistics. https://doi.org/10.18653/v1/p19-1355.

———. 2019b. "Energy and Policy Considerations for Deep Learning in NLP." *arXiv Preprint arXiv:1906.02243*, June, 3645–50. https://doi.org/10.18653/v1/p19-1355.

Sudhakar, Soumya, Vivienne Sze, and Sertac Karaman. 2023. "Data Centers on Wheels: Emissions from Computing Onboard Autonomous Vehicles." *IEEE Micro* 43 (1): 29–39. https://doi.org/10.1109/mm.2022.3219803.

Sullivan, Gary J., Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. 2012. "Overview of the High Efficiency Video Coding (HEVC) Standard." *IEEE*

Transactions on Circuits and Systems for Video Technology 22 (12): 1649–68. https://doi.org/10.1109/tcsvt.2012.2221191.

Sun, Siqi, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019. "Patient Knowledge Distillation for BERT Model Compression." In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/d19-1441.

Systems, Cerebras. 2021a. "The Wafer-Scale Engine 2: Scaling AI Compute Beyond GPUs." *Cerebras White Paper*. https://cerebras.ai/product-chip/.

———. 2021b. "Wafer-Scale Deep Learning Acceleration with the Cerebras CS-2." *Cerebras Technical Paper*.

Sze, Vivienne, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2017a. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." *Proceedings of the IEEE* 105 (12): 2295–2329. https://doi.org/10.1109/jproc.2017.2761740.

Sze, Vivienne, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017b. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." *Proceedings of the IEEE* 105 (12): 2295–2329. https://doi.org/10.1109/jproc.2017.2761740.

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013a. "Intriguing Properties of Neural Networks." *ICLR*, December. http://arxiv.org/abs/1312.6199v4.

———. 2013b. "Intriguing Properties of Neural Networks." Edited by Yoshua Bengio and Yann LeCun, December. http://arxiv.org/abs/1312.6199v4.

Tambe, Thierry, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. 2020. "Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference." In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 1–6. IEEE; IEEE. https://doi.org/10.1109/dac18072.2020.9218516.

Tan, Mingxing, and Quoc V Le. 2019a. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." In *International Conference on Machine Learning (ICML)*, 6105–14.

Tan, Mingxing, and Quoc V. Le. 2019b. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." In *Proceedings of the International Conference on Machine Learning (ICML)*, 6105–14.

———. 2019c. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." In *International Conference on Machine Learning*.

Team, The Theano Development, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, et al. 2016. "Theano: A Python Framework for Fast Computation of Mathematical Expressions," May. http://arxiv.org/abs/1605.02688v1.

Teerapittayanon, Surat, Bradley McDanel, and H. T. Kung. 2017. "BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks." *arXiv Preprint arXiv:1709.01686*, September. http://arxiv.org/abs/1709.01686v1.

The Sustainable Development Goals Report 2018. 2018. New York: United Nations. https://doi.org/10.18356/7d014b41-en.

Thompson, Neil, Tobias Spanuth, and Hyrum Anderson Matthews. 2023. "The Computational Limits of Deep Learning and the Future of AI." *Communications of the ACM* 66 (3): 48–57. https://doi.org/10.1145/3580309.

Thornton, James E. 1965. "Design of a Computer: The Control Data 6600." *Communications of the ACM* 8 (6): 330–35.

Thyagarajan, Aditya, Elías Snorrason, Curtis G. Northcutt, and Jonas Mueller 0001. 2022. "Identifying Incorrect Annotations in Multi-Label Classification Data." *CoRR*. https://doi.org/10.48550/ARXIV.2211.13895.

Tianqi, Chen et al. 2018. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 578–94.

Tirtalistyani, Rose, Murtiningrum Murtiningrum, and Rameshwar S. Kanwar. 2022. "Indonesia Rice Irrigation System: Time for Innovation." *Sustainability* 14 (19): 12477. https://doi.org/10.3390/su141912477.

Tramèr, Florian, Pascal Dupré, Gili Rusak, Giancarlo Pellegrino, and Dan Boneh. 2019. "AdVersarial: Perceptual Ad Blocking Meets Adversarial Machine Learning." In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2005–21. ACM. https://doi.org/10.1145/3319535.3354222.

Tramèr, Florian, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. "Stealing Machine Learning Models via Prediction APIs." In *25th USENIX Security Symposium (USENIX Security 16)*, 601–18.

Tsai, Min-Jen, Ping-Yi Lin, and Ming-En Lee. 2023. "Adversarial Attacks on Medical Image Classification." *Cancers* 15 (17): 4228. https://doi.org/10.3390/cancers15174228.

Tsai, Timothy, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen W. Keckler. 2021. "NVBitFI: Dynamic Fault Injection for GPUs." In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 284–91. IEEE; IEEE. https://doi.org/10.1109/dsn48987.2021.00041.

Tschand, Arya, Arun Tejusve Raghunath Rajan, Sachin Idgunji, Anirban Ghosh, Jeremy Holleman, Csaba Kiraly, Pawan Ambalkar, et al. 2024. "MLPerf Power: Benchmarking the Energy Efficiency of Machine Learning Systems from Microwatts to Megawatts for Sustainable AI." *arXiv Preprint arXiv:2410.12032*, October. http://arxiv.org/abs/2410.12032v2.

Uchida, Yusuke, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. 2017. "Embedding Watermarks into Deep Neural Networks." In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, 269–77. ACM; ACM. https://doi.org/10.1145/3078971.3078974.

Umuroglu, Yaman, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference." In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 65–74. ACM. https://doi.org/10.1145/3020078.3021744.

Un, and World Economic Forum. 2019. *A New Circular Vision for Electronics, Time for a Global Reboot*. PACE - Platform for Accelerating the Circular Economy. https://www3.weforum.org/docs/WEF/_A/_New/_Circular/_Vision/_for/_Electronics.pdf.

V. Forti, R. Kuehr, C. P. Baldé. 2020. *The Global e-Waste Monitor 2020: Quantities, Flows, and Circular Economy Potential*. United Nations University, International Telecommunication Union,; International Solid Waste Association. https://ewastemonitor.info.

Vangal, Sriram, Somnath Paul, Steven Hsu, Amit Agarwal, Saurabh Kumar, Ram Krishnamurthy, Harish Krishnamurthy, James Tschanz, Vivek De, and Chris H. Kim. 2021. "Wide-Range Many-Core SoC Design in Scaled CMOS: Challenges and Opportunities." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29 (5): 843–56. https://doi.org/10.1109/tvlsi.2021.3061649.

Vanschoren, Joaquin. 2018. "Meta-Learning: A Survey." *ArXiv Preprint arXiv:1810.03548*, October. http://arxiv.org/abs/1810.03548v1.

Velazco, Raoul, Gilles Foucard, and Paul Peronnard. 2010. "Combining Results of Accelerated Radiation Tests and Fault Injections to Predict the Error Rate of an Application Implemented in SRAM-Based FPGAs." *IEEE Transactions on Nuclear Science* 57 (6): 3500–3505. https://doi.org/10.1109/tns.2010.2087355.

Verma, Team Dual_Boot: Swapnil. 2022. "Elephant AI." *Hackster.io*. https://www.hackster.io/dual//_boot/elephant-ai-ba71e9.

Vinuesa, Ricardo, Hossein Azizpour, Iolanda Leite, Madeline Balaam, Virginia Dignum, Sami Domisch, Anna Felländer, Simone Daniela Langhans, Max Tegmark, and Francesco Fuso Nerini. 2020. "The Role of Artificial Intelligence in Achieving the Sustainable Development Goals." *Nature Communications* 11 (1): 233. https://doi.org/10.1038/s41467-019-14108-y.

Wachter, Sandra, Brent Mittelstadt, and Chris Russell. 2017. "Counterfactual Explanations Without Opening the Black Box: Automated Decisions and the GDPR." *SSRN Electronic Journal* 31: 841. https://doi.org/10.2139/ssrn.3063289.

Wald, Peter H., and Jeffrey R. Jones. 1987. "Semiconductor Manufacturing: An Introduction to Processes and Hazards." *American Journal of Industrial Medicine* 11 (2): 203–21. https://doi.org/10.1002/ajim.4700110209.

Wan, Zishen, Aqeel Anwar, Yu-Shun Hsiao, Tianyu Jia, Vijay Janapa Reddi, and Arijit Raychowdhury. 2021. "Analyzing and Improving Fault Tolerance of Learning-Based Navigation Systems." In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 841–46. IEEE; IEEE. https://doi.org/10.1109/dac18074.2021.9586116.

Wan, Zishen, Yiming Gan, Bo Yu, S Liu, A Raychowdhury, and Y Zhu. 2023. "Vpp: The Vulnerability-Proportional Protection Paradigm Towards Reliable Autonomous Machines." In *Proceedings of the 5th International Workshop on Domain Specific System Architecture (DOSSA)*, 1–6.

Wang, Alex, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. "SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems." *arXiv Preprint arXiv:1905.00537*, May. http://arxiv.org/abs/1905.00537v3.

Wang, Alex, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding." *arXiv Preprint arXiv:1804.07461*, April. http://arxiv.org/abs/1804.07461v3.

Wang, Tianlu, Jieyu Zhao, Mark Yatskar, Kai-Wei Chang, and Vicente Ordonez. 2019. "Balanced Datasets Are Not Enough: Estimating and Mitigating Gender Bias in Deep Image Representations." In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 5309–18. IEEE. https://doi.org/10.1109/iccv.2019.00541.

Wang, Xin, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. 2018. "SkipNet: Learning Dynamic Routing in Convolutional Networks." In *Computer Vision – ECCV 2018*, 420–36. Springer; Springer International Publishing. https://doi.org/10.1007/978-3-030-01261-8/_25.

Wang, Yaqing, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2020. "Generalizing from a Few Examples: A Survey on Few-Shot Learning." *ACM Computing Surveys* 53 (3): 1–34. https://doi.org/10.1145/3386252.

Wang, Y., and P. Kanwar. 2019. "BFloat16: The Secret to High Performance on Cloud TPUs." *Google Cloud Blog*.

Wang, Yu Emma, Gu-Yeon Wei, and David Brooks. 2019. "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning." *arXiv Preprint arXiv:1907.10701*, July. http://arxiv.org/abs/1907.10701v4.

Warden, Pete. 2018. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition." *arXiv Preprint arXiv:1804.03209*, April. http://arxiv.org/abs/1804.03209v1.

Weicker, Reinhold P. 1984. "Dhrystone: A Synthetic Systems Programming Benchmark." *Communications of the ACM* 27 (10): 1013–30. https://doi.org/10.1145/358274.358283.

Werchniak, Andrew, Roberto Barra Chicote, Yuriy Mishchenko, Jasha Droppo, Jeff Condal, Peng Liu, and Anish Shah. 2021. "Exploring the Application of Synthetic Audio in Training Keyword Spotters." In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 7993–96. IEEE; IEEE. https://doi.org/10.1109/icassp39728.2021.9413448.

Wiener, Norbert. 1960. "Some Moral and Technical Consequences of Automation: As Machines Learn They May Develop Unforeseen Strategies at Rates That Baffle Their Programmers." *Science* 131 (3410): 1355–58. https://doi.org/10.1126/science.131.3410.1355.

Wilkening, Mark, Vilas Sridharan, Si Li, Fritz Previlon, Sudhanva Gurumurthi, and David R. Kaeli. 2014. "Calculating Architectural Vulnerability Factors for Spatial Multi-Bit Transient Faults." In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 293–305. IEEE; IEEE. https://doi.org/10.1109/micro.2014.15.

Witten, Ian H., and Eibe Frank. 2002. "Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations." *ACM SIGMOD Record* 31 (1): 76–77. https://doi.org/10.1145/507338.507355.

Wolpert, D. H., and W. G. Macready. 1997. "No Free Lunch Theorems for Optimization." *IEEE Transactions on Evolutionary Computation* 1 (1): 67–82. https://doi.org/10.1109/4235.585893.

Wu, Bichen, Kurt Keutzer, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, and Yangqing Jia. 2019. "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search." In *2019 IEEE/CVF Conference on Computer Vision and*

Pattern Recognition (CVPR), 10726–34. IEEE. https://doi.org/10.1109/cvpr.2019.01099.

Wu, Carole-Jean, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, et al. 2019. "Machine Learning at Facebook: Understanding Inference at the Edge." In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 331–44. IEEE; IEEE. https://doi.org/10.1109/hpca.2019.00048.

Wu, Carole-Jean, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, et al. 2022. "Sustainable Ai: Environmental Implications, Challenges and Opportunities." *Proceedings of Machine Learning and Systems* 4: 795–813.

Wu, Hao, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. "Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation." *arXiv Preprint arXiv:2004.09602* abs/2004.09602 (April). http://arxiv.org/abs/2004.09602v1.

Wu, Jian, Hao Cheng, and Yifan Zhang. 2019. "Fast Neural Networks: Efficient and Adaptive Computation for Inference." In *Advances in Neural Information Processing Systems*.

Wu, Jiaxiang, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. "Quantized Convolutional Neural Networks for Mobile Devices." In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 4820–28. IEEE. https://doi.org/10.1109/cvpr.2016.521.

Xin, Ji, Raphael Tang, Yaoliang Yu, and Jimmy Lin. 2021. "BERxiT: Early Exiting for BERT with Better Fine-Tuning and Extension to Regression." In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, edited by Paola Merlo, Jorg Tiedemann, and Reut Tsarfaty, 91–104. Online: Association for Computational Linguistics. https://doi.org/10.18653/v1/2021.eacl-main.8.

Xingyu, Huang et al. 2019. "Addressing the Memory Bottleneck in AI Accelerators." *IEEE Micro*.

Xu, Ruijie, Zengzhi Wang, Run-Ze Fan, and Pengfei Liu. 2024. "Benchmarking Benchmark Leakage in Large Language Models." *arXiv Preprint arXiv:2404.18824*, April. http://arxiv.org/abs/2404.18824v1.

Xu, Xiaolong, Fan Li, Wei Zhang, Liang He, and Ruidong Li. 2021. "Edge Intelligence: Architectures, Challenges, and Applications." *IEEE Internet of Things Journal* 8 (6): 4229–49.

Yang, Le, Yizeng Han, Xi Chen, Shiji Song, Jifeng Dai, and Gao Huang. 2020. "Resolution Adaptive Networks for Efficient Inference." In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2366–75. IEEE. https://doi.org/10.1109/cvpr42600.2020.00244.

Yao, Zhewei, Amir Gholami, Sheng Shen, Kurt Keutzer, and Michael W. Mahoney. 2021. "HAWQ-V3: Dyadic Neural Network Quantization." In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, 11875–86. PMLR.

Yeh, Y. C. n.d. "Triple-Triple Redundant 777 Primary Flight Computer." In *1996 IEEE Aerospace Applications Conference. Proceedings*, 1:293–307. IEEE; IEEE. https://doi.org/10.1109/aero.1996.495891.

Yosinski, Jason, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. "How Transferable Are Features in Deep Neural Networks?" *Advances in Neural Information Processing Systems* 27.

You, Jie, Jae-Won Chung, and Mosharaf Chowdhury. 2023. "Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training." In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 119–39. Boston, MA: USENIX Association. https://www.usenix.org/conference/nsdi23/presentation/you.

Yu, Jun, Peng Li, and Zhenhua Wang. 2023. "Efficient Early Exiting Strategies for Neural Network Acceleration." *IEEE Transactions on Neural Networks and Learning Systems*.

Zafrir, Ofir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. "Q8BERT: Quantized 8Bit BERT." In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, 36–39. IEEE; IEEE. https://doi.org/10.1109/emc2-nips53020.2019.00016.

Zaharia, Matei, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Corey Murching, et al. 2018. "Accelerating the Machine Learning Lifecycle with MLflow." *Databricks*.

Zeghidour, Neil, Olivier Teboul, Félix de Chaumont Quitry, and Marco Tagliasacchi. 2021. "LEAF: A Learnable Frontend for Audio Classification." *arXiv Preprint arXiv:2101.08596*, January. http://arxiv.org/abs/2101.08596v1.

Zhan, Ruiting, Zachary Oldenburg, and Lei Pan. 2018. "Recovery of Active Cathode Materials from Lithium-Ion Batteries Using Froth Flotation." *Sustainable Materials and Technologies* 17 (September): e00062. https://doi.org/10.1016/j.susmat.2018.e00062.

Zhang, Chengliang, Minchen Yu, Wei Wang 0030, and Feng Yan 0001. 2019. "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving." In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 1049–62. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang.

Zhang, Jeff Jun, Tianyu Gu, Kanad Basu, and Siddharth Garg. 2018. "Analyzing and Mitigating the Impact of Permanent Faults on a Systolic Array Based Neural Network Accelerator." In *2018 IEEE 36th VLSI Test Symposium (VTS)*, 1–6. IEEE; IEEE. https://doi.org/10.1109/vts.2018.8368656.

Zhang, Jeff, Kartheek Rangineni, Zahra Ghodsi, and Siddharth Garg. 2018. "ThUnderVolt: Enabling Aggressive Voltage Underscaling and Timing Error Resilience for Energy Efficient Deep Learning Accelerators." In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 1–6. IEEE. https://doi.org/10.1109/dac.2018.8465918.

Zhang, Qingxue, Dian Zhou, and Xuan Zeng. 2017. "Highly Wearable Cuff-Less Blood Pressure and Heart Rate Monitoring with Single-Arm Electrocardiogram and Photoplethysmogram Signals." *BioMedical Engineering OnLine* 16 (1): 23. https://doi.org/10.1186/s12938-017-0317-z.

Zhang, Xitong, Jialin Song, and Dacheng Tao. 2020. "Efficient Task-Specific Adaptation for Deep Models." In *International Conference on Learning Representations (ICLR)*.

Zhang, Yi, Jianlei Yang, Linghao Song, Yiyu Shi, Yu Wang, and Yuan Xie. 2021. "Learning-Based Efficient Sparsity and Quantization for Neural Network

Compression." *IEEE Transactions on Neural Networks and Learning Systems* 32 (9): 3980–94.

Zhang, Y., J. Li, and H. Ouyang. 2020. "Optimizing Memory Access for Deep Learning Workloads." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39 (11): 2345–58.

Zhao, Jiawei, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. 2024. "GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection," March. http://arxiv.org/abs/2403.03507v2.

Zhao, Mark, and G. Edward Suh. 2018. "FPGA-Based Remote Power Side-Channel Attacks." In *2018 IEEE Symposium on Security and Privacy (SP)*, 229–44. IEEE; IEEE. https://doi.org/10.1109/sp.2018.00049.

Zhao, Yue, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. 2018. "Federated Learning with Non-IID Data." *CoRR* abs/1806.00582 (June). http://arxiv.org/abs/1806.00582v2.

Zheng, Lianmin, Ziheng Jia, Yida Gao, Jiacheng Lin, Song Han, Xuehai Geng, Eric Zhao, and Tianqi Wu. 2020. "Ansor: Generating High-Performance Tensor Programs for Deep Learning." *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 863–79.

Zhou, Aojun, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. "Learning n:m Fine-Grained Structured Sparse Neural Networks from Scratch," February. http://arxiv.org/abs/2102.04010v2.

Zhu, Chenzhuo, Song Han, Huizi Mao, and William J. Dally. 2017. "Trained Ternary Quantization." *International Conference on Learning Representations (ICLR)*.

Zoph, Barret, and Quoc V Le. 2017a. "Neural Architecture Search with Reinforcement Learning." In *International Conference on Learning Representations (ICLR)*.

Zoph, Barret, and Quoc V. Le. 2017b. "Neural Architecture Search with Reinforcement Learning." In *International Conference on Learning Representations*.