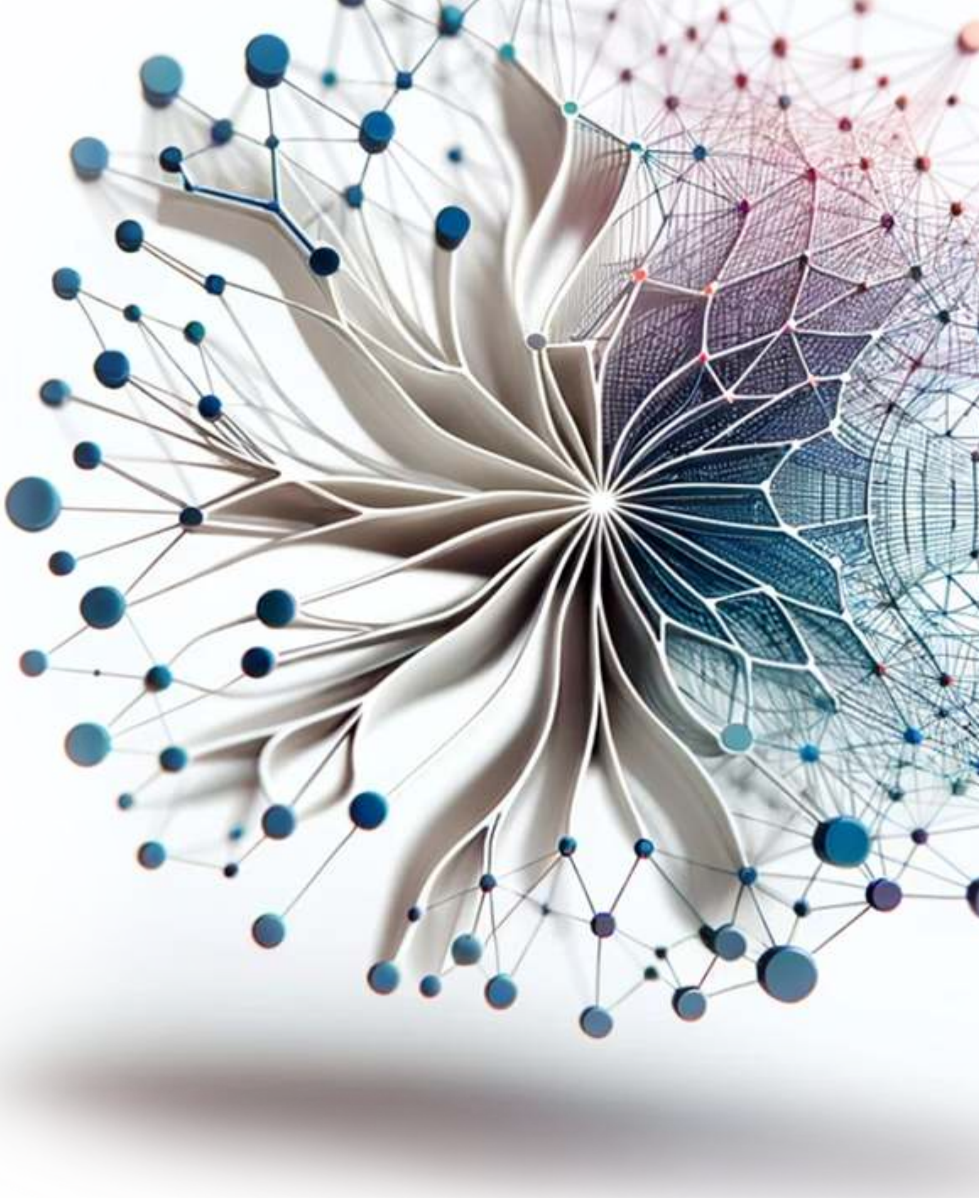


Machine Learning Systems

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.

Table of contents

Preface

Welcome to Machine Learning Systems. This book is your gateway to the fast-paced world of AI systems. It is an extension of the course CS249r at Harvard University.

We have created this open-source book as a collaborative effort to bring together insights from students, professionals, and the broader community of AI practitioners. Our goal is to develop a comprehensive guide that explores the intricacies of AI systems and their numerous applications.

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

This isn't a static textbook; it's a living, breathing document. We're making it open-source and continuously updated to meet the ever-changing needs of this dynamic field. Expect a rich blend of expert knowledge that guides you through the complex interplay between cutting-edge algorithms and the foundational principles that make them work. We're setting the stage for the next big leap in AI innovation.

Why We Wrote This Book

We're in an age where technology is always evolving. Open collaboration and sharing knowledge are the building blocks of true innovation. That's the spirit behind this effort. We go beyond the traditional textbook model to create a living knowledge hub, so that we can all share and learn from one another.

The book focuses on AI systems' principles and case studies, aiming to give you a deep understanding that will help you navigate the ever-changing landscape of AI systems. By keeping it open, we're not just making learning accessible but inviting new ideas and ongoing improvements. In short, we're building a community where knowledge is free to grow and light the way forward in global AI technology.

What You'll Need to Know

To dive into this book, you don't need to be an AI expert. All you need is a basic understanding of computer science concepts and a curiosity to explore how AI systems work. This is where innovation happens, and a basic grasp of programming and data structures will be your compass.

Content Transparency Statement

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 and editorial oversight to make sure the content is accurate and relevant. However, no one is perfect, so inaccuracies may still exist. We highly value your feedback and encourage you to provide corrections or suggestions. This collaborative approach is crucial for enhancing and maintaining the quality of the content contained within and making high-quality information globally accessible.

Want to Help Out?

If you're interested in contributing, you can find the guidelines here.

Get in Touch

Do you have questions or feedback? Feel free to e-mail Prof. Vijay Janapa Reddi directly, or you are welcome to start a discussion thread on GitHub.

Contributors

A big thanks to everyone who's helped make this book what it is! You can see the full list of individual contributors here and additional GitHub style details here. Join us as a contributor!

Copyright

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 practioners. 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

We are grateful for the support from various funding agencies and companies that backed the teaching assistants involved in this work. The following organizations played a crucial role in bringing this project to life:

HDSI | Harvard Data Science Initiative

HARVARD Extension School

Google

NSF

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
Naeem Khoshnevis
Marcelo Rovai
Sara Khosravi
Douwe den Blanken
shanzehbatool
Kai Kleinbard
Elias Nuwara
Matthew Stewart
Jared Ping
Itai Shapira
Maximilian Lam
Jayson Lin
Sophia Cho
Andrea
Jeffrey Ma
Alex Rodriguez
Korneel Van den Berghe
Colby Banbury
Zishen Wan
Abdulrahman Mahmoud
Srivatsan Krishnan
Divya Amirtharaj
Emeka Ezike
Aghyad Deeb
Haoran Qiu
marin-llobet
Emil Njor
Aditi Raju

Jared Ni
Michael Schnebly
oishib
ELSuitorHarvard
Henry Bae
Jae-Won Chung
Yu-Shun Hsiao
Mark Mazumder
Marco Zennaro
Eura Nofshin
Andrew Bass
Pong Trairatvorakul
Jennifer Zhou
Shvetank Prakash
Alex Oesterling
Arya Tschand
Bruno Scaglione
Gauri Jain
Allen-Kuang
Fin Amin
Fatima Shah
The Random DIY
gnodipac886
Sercan Aygün
Baldassarre Cesarano
Abenezer
Bilge Acun
yanjingl
Yang Zhou
abigailswallow
Jason Yik
happyappledog
Curren Iyer
Emmanuel Rassou
Sonia Murthy
Shreya Johri
Jessica Quaye
Vijay Edupuganti
Costin-Andrei Oncescu
Annie Laurie Cook
Jothi Ramaswamy
Batur Arslan
Fatima Shah

a-saraf
songhan
Zishen

About the Book

Overview

Welcome to this collaborative textbook, developed as part of the CS249r Machine Learning Systems class at Harvard University. Our goal is to provide a comprehensive resource for educators and students seeking to understand machine learning systems. This book is continually updated to incorporate the latest insights and effective teaching strategies.

What's Inside the Book

We explore the technical foundations of machine learning systems, the challenges of building and deploying these systems across the computing continuum, and the vast array of applications they enable. A unique aspect of this book is its function as a conduit to seminal scholarly works and academic research papers, aimed at enriching the reader's understanding and encouraging deeper exploration of the subject. This approach seeks to bridge the gap between pedagogical materials and cutting-edge research trends, offering a comprehensive guide that is in step with the evolving field of applied machine learning.

To improve the learning experience, we have included a variety of supplementary materials. Throughout the book, you will find slides that summarize key concepts, videos that provide in-depth explanations and demonstrations, exercises that reinforce your understanding, and labs that offer hands-on experience with the tools and techniques discussed. These additional resources are designed to cater to different learning styles and help you gain a deeper, more practical understanding of the subject matter.

Topics Explored

This textbook offers a comprehensive exploration of various aspects of machine learning systems, covering the entire end-to-end workflow. Starting with foundational concepts, it progresses through essential areas such as data engineering, AI frameworks, and model training.

To enhance the learning experience, we included a diverse array of supplementary materials. These resources consist of slides that summarize key concepts, videos providing detailed explanations and demonstrations, exercises designed to reinforce understanding, and labs that offer hands-on experience with the discussed tools and techniques.

Readers will gain insights into optimizing models for efficiency, deploying AI across different hardware platforms, and benchmarking performance. The book also delves into advanced topics, including security, privacy, responsible and sustainable AI, robust AI, and generative AI. Additionally, it examines the social impact of AI, concluding with an emphasis on the positive contributions AI can make to society.

Key Learning Outcomes

Readers will acquire skills in training and deploying deep neural network models on various platforms, along with understanding the broader challenges involved in their design, development, and deployment. Specifically, after completing this book, learners will be able to:

Tip

1. Explain core concepts and their relevance to AI systems.
2. Describe the fundamental components and architecture of AI systems.
3. Compare and contrast various hardware platforms for AI deployment, selecting appropriate options for specific use cases.
4. Design and implement training processes for AI models across different systems.
5. Apply optimization techniques to improve AI model performance and efficiency.

6. Analyze real-world AI applications and their implementation strategies.
7. Evaluate current challenges in AI systems and predict future trends in the field.
8. Develop a complete machine learning-enabled project, from conception to deployment.
9. Troubleshoot common issues in AI model training and deployment.
10. Critically assess the ethical implications and societal impacts of AI systems.

Prerequisites for Readers

- **Basic Programming Skills:** We recommend that you have some prior programming experience, ideally in Python. A grasp of variables, data types, and control structures will make it easier to engage with the book.
- **Some Machine Learning Knowledge:** While not mandatory, a basic understanding of machine learning concepts will help you absorb the material more readily. If you're new to the field, the book provides enough background information to get you up to speed.
- **Basic Systems Knowledge:** A basic level of systems knowledge at an undergraduate junior or senior level is recommended. Understanding system architecture, operating systems, and basic networking will be beneficial.
- **Python Programming (Optional):** If you're familiar with Python, you'll find it easier to engage with the coding sections of the book. Knowing libraries like NumPy, scikit-learn, and TensorFlow will be particularly helpful.
- **Willingness to Learn:** The book is designed to be accessible to a broad audience, with varying levels of technical expertise. A willingness to challenge yourself and engage in practical exercises will help you get the most out of it.
- **Resource Availability:** For the hands-on aspects, you'll need a computer with Python and the relevant libraries installed.

Optional access to development boards or specific hardware will also be beneficial for experimenting with machine learning model deployment.

By meeting these prerequisites, you'll be well-positioned to deepen your understanding of machine learning systems, engage in coding exercises, and even implement practical applications on various devices.

Who Should Read This

This book is designed for individuals at different stages of their journey with machine learning systems, from beginners to those more advanced in the field. It introduces fundamental concepts and progresses to complex topics relevant to the machine learning community and expansive research areas. The key audiences for this book include:

- **Students in Computer Science and Electrical Engineering:** Senior and graduate students will find this book particularly valuable. It introduces the techniques essential for designing and building ML systems, focusing on foundational knowledge rather than exhaustive detail—often the focus of classroom instruction. This book will provide the necessary background and context, enabling instructors to explore advanced topics more deeply. An essential feature is its end-to-end perspective, which is often overlooked in traditional curricula.

- **Systems Engineers:** This book serves as a guide for engineers seeking to understand the complexities of intelligent systems and applications, particularly involving ML. It encompasses the conceptual frameworks and practical components that comprise an ML system, extending beyond the specific areas you might encounter in your professional role.

- **Researchers and Academics:** For researchers, this book addresses the distinct challenges of executing machine learning algorithms across diverse platforms. As efficiency gains importance, a robust understanding of systems, beyond algorithms alone, is crucial for developing more efficient models. The book references seminal papers, directing researchers to works that have influenced the field and establishing connections between various areas with significant implications for their research.

How to Navigate This Book

To get the most out of this book, we recommend a structured learning approach that leverages the various resources provided. Each chapter includes slides, videos, exercises, and labs to cater to different learning styles and reinforce your understanding.

1. **Fundamentals (Chapters 1-3):** Start by building a strong foundation with the initial chapters, which provide an introduction to AI and cover core topics like AI systems and deep learning.
2. **Workflow (Chapters 4-6):** With that foundation, move on to the chapters focused on practical aspects of the AI model building process like workflows, data engineering, and frameworks.
3. **Training (Chapters 7-10):** These chapters offer insights into effectively training AI models, including techniques for efficiency, optimizations, and acceleration.
4. **Deployment (Chapters 11-13):** Learn about deploying AI on devices and monitoring the operationalization through methods like benchmarking, on-device learning, and MLOps.
5. **Advanced Topics (Chapters 14-18):** Critically examine topics like security, privacy, ethics, sustainability, robustness, and generative AI.
6. **Social Impact (Chapter 19):** Explore the positive applications and potential of AI for societal good.
7. **Conclusion (Chapter 20):** Reflect on the key takeaways and future directions in AI systems.

While the book is designed for progressive learning, we encourage an interconnected learning approach that allows you to navigate chapters based on your interests and needs. Throughout the book, you'll find case studies and hands-on exercises that help you relate theory to real-world applications. We also recommend participating in forums and groups to engage in discussions, debate concepts, and share insights with fellow learners. Regularly revisiting chapters can help reinforce your learning and offer new perspectives on the concepts covered. By adopting this structured yet flexible approach and actively engaging with the content and the community, you'll embark on a fulfilling and enriching learning experience that maximizes your understanding.

Chapter-by-Chapter Insights

Here's a closer look at what each chapter covers. We have structured the book into six main sections: Fundamentals, Workflow, Training, Deployment, Advanced Topics, and Impact. These sections closely reflect the major components of a typical machine learning pipeline, from understanding the basic concepts to deploying and maintaining AI systems in real-world applications. By organizing the content in this manner, we aim to provide a logical progression that mirrors the actual process of developing and implementing AI systems.

Fundamentals

In the Fundamentals section, we lay the groundwork for understanding AI. This is far from being a thorough deep dive into the algorithms, but we aim to introduce key concepts, provide an overview of machine learning systems, and dive into the principles and algorithms of deep learning that power AI applications in their associated systems. This section equips you with the essential knowledge needed to grasp the subsequent chapters.

1. **Introduction:** This chapter sets the stage, providing an overview of AI and laying the groundwork for the chapters that follow.
2. **ML Systems:** We introduce the basics of machine learning systems, the platforms where AI algorithms are widely applied.
3. **Deep Learning Primer:** This chapter offers a brief introduction to the algorithms and principles that underpin AI applications in ML systems.

Workflow

The Workflow section guides you through the practical aspects of building AI models. We break down the AI workflow, discuss data engineering best practices, and review popular AI frameworks. By the end of this section, you'll have a clear understanding of the steps involved in developing proficient AI applications and the tools available to streamline the process.

4. **AI Workflow:** This chapter breaks down the machine learning workflow, offering insights into the steps leading to proficient AI applications.
5. **Data Engineering:** We focus on the importance of data in AI systems, discussing how to effectively manage and organize data.

6. **AI Frameworks:** This chapter reviews different frameworks for developing machine learning models, guiding you in choosing the most suitable one for your projects.

Training

In the Training section, we explore techniques for training efficient and reliable AI models. We cover strategies for achieving efficiency, model optimizations, and the role of specialized hardware in AI acceleration. This section empowers you with the knowledge to develop high-performing models that can be seamlessly integrated into AI systems.

7. **AI Training:** This chapter explores model training, exploring techniques for developing efficient and reliable models.
8. **Efficient AI:** Here, we discuss strategies for achieving efficiency in AI applications, from computational resource optimization to performance enhancement.

9. **Model Optimizations:** We explore various avenues for optimizing AI models for seamless integration into AI systems.
10. **AI Acceleration:** We discuss the role of specialized hardware in enhancing the performance of AI systems.

Deployment

The Deployment section focuses on the challenges and solutions for deploying AI models. We discuss benchmarking methods to evaluate AI system performance, techniques for on-device learning to improve efficiency and privacy, and the processes involved in ML operations. This section equips you with the skills to effectively deploy and maintain AI functionalities in AI systems.

11. **Benchmarking AI:** This chapter focuses on how to evaluate AI systems through systematic benchmarking methods.
12. **On-Device Learning:** We explore techniques for localized learning, which enhances both efficiency and privacy.
13. **ML Operations:** This chapter looks at the processes involved in the seamless integration, monitoring, and maintenance of AI functionalities.

Advanced Topics

In the Advanced Topics section, We will study the critical issues surrounding AI. We address privacy and security concerns, explore the ethical principles of responsible AI, discuss strategies for sustainable AI development, examine techniques for building robust AI models, and introduce the exciting field of generative AI. This section broadens your understanding of the complex landscape of AI and prepares you to navigate its challenges.

14. **Security & Privacy:** As AI becomes more ubiquitous, this chapter addresses the crucial aspects of privacy and security in AI systems.
15. **Responsible AI:** We discuss the ethical principles guiding the responsible use of AI, focusing on fairness, accountability, and transparency.
16. **Sustainable AI:** This chapter explores practices and strategies for sustainable AI, ensuring long-term viability and reduced environmental impact.
17. **Robust AI:** We discuss techniques for developing reliable and robust AI models that can perform consistently across various conditions.
18. **Generative AI:** This chapter explores the algorithms and techniques behind generative AI, opening avenues for innovation and creativity.

Social Impact

The Impact section highlights the transformative potential of AI in various domains. We showcase real-world applications of TinyML in healthcare, agriculture, conservation, and other areas where AI is making a positive difference. This section inspires you to leverage the power of AI for societal good and to contribute to the development of impactful solutions.

19. **AI for Good:** We highlight positive applications of TinyML in areas like healthcare, agriculture, and conservation.

Closing

In the Closing section, we reflect on the key learnings from the book and look ahead to the future of AI. We synthesize the concepts covered, discuss emerging trends, and provide guidance on continuing your learning journey in this rapidly evolving field. This section leaves you with a comprehensive understanding of AI and the excitement to apply your knowledge in innovative ways.

20. **Conclusion:** The book concludes with a reflection on the key learnings and future directions in the field of AI.

Tailored Learning

We understand that readers have diverse interests; some may wish to grasp the fundamentals, while others are eager to delve into advanced topics like hardware acceleration or AI ethics. To help you navigate the book more effectively, we've created a persona-based reading guide tailored to your specific interests and goals. This guide assists you in identifying the reader persona that best matches your interests. Each persona represents a distinct reader profile with specific objectives. By selecting the persona that resonates with you, you can focus on the chapters and sections most relevant to your needs.

Persona	Description	Chapters	Focus
The TinyML Newbie	You are new to the field of TinyML and eager to learn the basics.	1-3, 8, 9, 10, 12	Understand the fundamentals, gain insights into efficient and optimized ML, and learn about on-device learning.
The EdgeML Enthusiast	You have some TinyML knowledge and are interested in exploring the broader world of EdgeML.	1-3, 8, 9, 10, 12, 13	Build a strong foundation, delve into the intricacies of efficient ML, and explore the operational aspects of embedded systems.
The Computer Visionary	You are fascinated by computer vision and its applications in TinyML and EdgeML.	1-3, 5, 8-10, 12, 13, 17, 20	Start with the basics, explore data engineering, and study methods for optimizing ML models. Learn about robustness and the future of ML systems.
The Data Maestro	You are passionate about data and its crucial role in ML systems.	1-5, 8-13	Gain a comprehensive understanding of data's role in ML systems, explore the ML workflow, and dive into model optimization and deployment considerations.

Persona	Description	Chapters	Focus
The Hardware Hero	You are excited about the hardware aspects of ML systems and how they impact model performance.	1-3, 6, 8-10, 12, 14, 17, 20	Build a solid foundation in ML systems and frameworks, explore challenges of optimizing models for efficiency, hardware-software co-design, and security aspects.
The Sustainability Champion	You are an advocate for sustainability and want to learn how to develop eco-friendly AI systems.	1-3, 8-10, 12, 15, 16, 20	Begin with the fundamentals of ML systems and TinyML, explore model optimization techniques, and learn about responsible and sustainable AI practices.
The AI Ethicist	You are concerned about the ethical implications of AI and want to ensure responsible development and deployment.	1-3, 5, 7, 12, 14-16, 19, 20	Gain insights into the ethical considerations surrounding AI, including fairness, privacy, sustainability, and responsible development practices.
The Full-Stack ML Engineer	You are a seasoned ML expert and want to deepen your understanding of the entire ML system stack.	The entire book	Understand the end-to-end process of building and deploying ML systems, from data engineering and model optimization to hardware acceleration and ethical considerations.

Join the Community

Learning in the fast-paced world of AI is a collaborative journey. We set out to nurture a vibrant community of learners, innovators, and contributors. As you explore the concepts and engage with the exercises, we encourage you to share your insights and experiences. Whether it's a novel approach, an interesting application, or a thought-provoking question, your contributions can enrich the learning ecosystem. Engage in discussions, offer and seek guidance, and collaborate on projects to foster a culture of mutual growth and learning. By sharing knowledge, you play an important role in fostering a globally connected, informed, and empowered community.

Chapter 1

Introduction

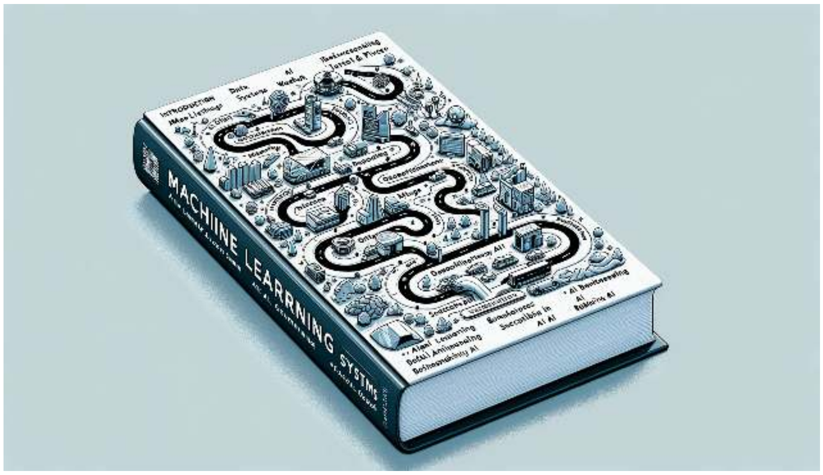


Figure 1.1: *DALL·E 3 Prompt: A detailed, rectangular, flat 2D illustration depicting a roadmap of a book's chapters on machine learning systems, set on a crisp, clean white background. The image features a winding road traveling through various symbolic landmarks. Each landmark represents a chapter 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 Why Machine Learning Systems Matter

AI is everywhere. Consider your morning routine: You wake up to an AI-powered smart alarm that learned your sleep patterns. Your phone suggests your route to work, having learned from traffic patterns. During your commute, your music app automatically creates a playlist it thinks you'll enjoy. At work, your email client filters spam and prioritizes important messages. Throughout the day, your smartwatch monitors your activity, suggesting when to move or exercise. In the evening, your streaming service recommends shows you might like, while your smart home devices adjust lighting and temperature based on your learned preferences.

But these everyday conveniences are just the beginning. AI is transforming our world in extraordinary ways. Today, AI systems detect early-stage cancers with unprecedented accuracy, predict and track extreme weather events to save lives, and accelerate drug discovery by simulating millions of molecular interactions. Autonomous vehicles navigate complex city streets while processing real-time sensor data from dozens of sources. Language models engage in sophisticated conversations, translate between hundreds of languages, and help scientists analyze vast research databases. In scientific laboratories, AI systems are making breakthrough discoveries - from predicting protein structures that unlock new medical treatments to identifying promising materials for next-generation solar cells and batteries. Even in creative fields, AI collaborates with artists and musicians to explore new forms of expression, pushing the boundaries of human creativity.

This isn't science fiction—it's the reality of how artificial intelligence, specifically machine learning systems, has become woven into the fabric of our daily lives. In the early 1990s, Mark Weiser, a pioneering computer scientist, introduced the world to a revolutionary concept that would forever change how we interact with technology. This vision was succinctly captured in his seminal paper, "The Computer for the 21st Century" (see Figure 1.2). Weiser envisioned a future where computing would be seamlessly integrated into our environments, becoming an invisible, integral part of daily life.

The Computer for the 21st Century

Specialized elements of hardware and software, connected by wires, radio waves and infrared, will be so ubiquitous that no one will notice their presence

by Mark Weiser

[illegible]

Figure 1.2: Ubiquitous computing as envisioned by Mark Weiser.

He termed this concept "ubiquitous computing," promising a world where technology would serve us without demanding our constant attention or interaction. Today, we find ourselves living in Weiser's envisioned future, largely enabled by machine learning systems. The true essence of his vision—creating an intelligent environment that can anticipate our needs and act on our behalf—has become reality through the development and deployment of ML systems that span entire ecosystems, from powerful cloud data centers to edge devices to the tiniest IoT sensors.

Yet most of us rarely think about the complex systems that make this possible. Behind each of these seemingly simple interactions lies a sophisticated infrastructure of data, algorithms, and computing resources working together. Understanding how these systems work—their capabilities, limitations, and requirements—has become increasingly critical as they become more integrated into our world.

To appreciate the magnitude of this transformation and the complexity of modern machine learning systems, we need to understand how we got here. The journey from early artificial intelligence to today's ubiquitous ML systems is a story of not just technological evolution, but of changing perspectives on what's possible and what's necessary to make AI practical and reliable.

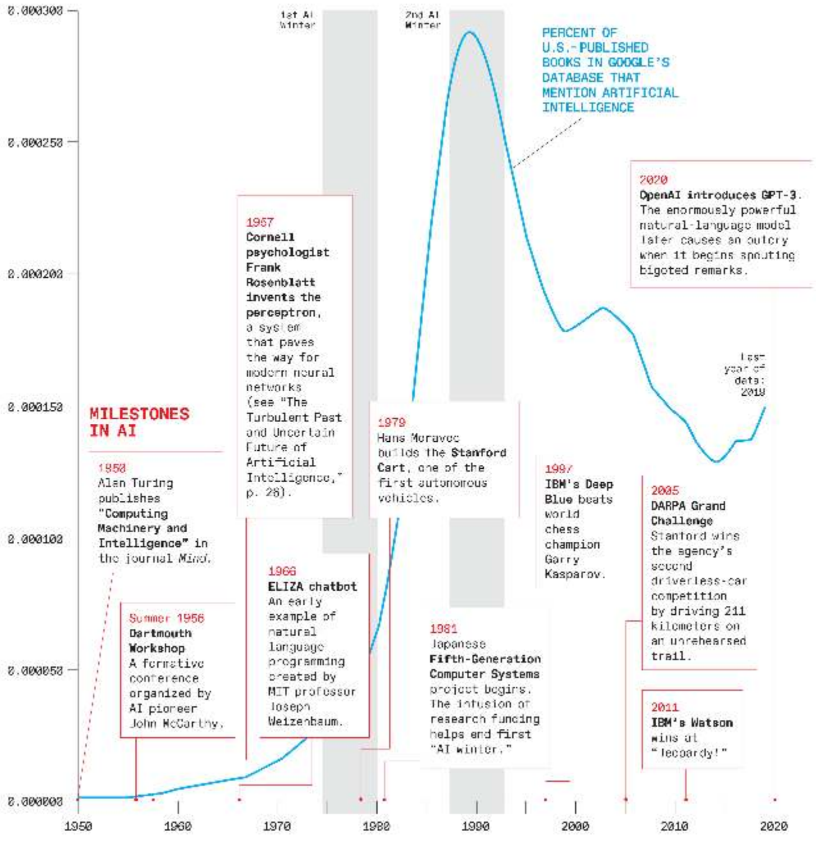
1.2 The Evolution of AI

The evolution of AI, depicted in the timeline shown in Figure 1.3, highlights key milestones such as the development of the **perceptron**[1] 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.

Let's explore how we got here.

1 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.

Figure 1.3: Milestones in AI from 1950 to 2020. Source: IEEE Spectrum



1950
Alan Turing publishes **"Computing Machinery and Intelligence"** in the journal *Mind*.

Summer 1956
Dartmouth Workshop
A formative conference organized by AI pioneer John McCarthy.

1957
Cornell psychologist Frank Rosenblatt invents the perceptron, a system that paves the way for modern neural networks (see "The Turbulent Past and Uncertain Future of Artificial Intelligence," p. 28).

1966
ELIZA chatbot
An early example of natural language programming created by MIT professor Joseph Weizenbaum.

1979
Hans Moravec builds the **Stanford Cart**, one of the first autonomous vehicles.

1981
Japanese **Fifth-Generation Computer Systems** project begins. The infusion of research funding helps end first "AI winter."

1997
IBM's Deep Blue beats world chess champion Garry Kasparov.

2005
DARPA Grand Challenge
Stanford wins the agency's second driverless-car competition by driving 211 kilometers on an unrehearsed trail.

2011
IBM's Watson wins at "Jeopardy!"

2020
OpenAI introduces GPT-3. The enormously powerful natural-language model later causes an outcry when it begins spouting bigoted remarks.

1.2.1 Symbolic AI (1956-1974)

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:

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 like 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.

1.2.2 Expert Systems(1970s-1980s)

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
    There is 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—adding new rules often conflicted with existing ones, and medical knowledge itself kept evolving. 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.

1.2.3 Statistical Learning: A Paradigm Shift (1990s)

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**[2] 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

2 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.

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)
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

Aspect	Symbolic AI	Expert Systems	Statistical Learning	Shallow / Deep Learning
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.

1.2.4 Shallow Learning (2000s)

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 - like 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 because: 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.

1.2.5 Deep Learning (2012-Present)

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, 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." 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—from simple to complex, 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 1.4, 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 achieved a 15.3% error rate, dramatically outperforming all existing methods.

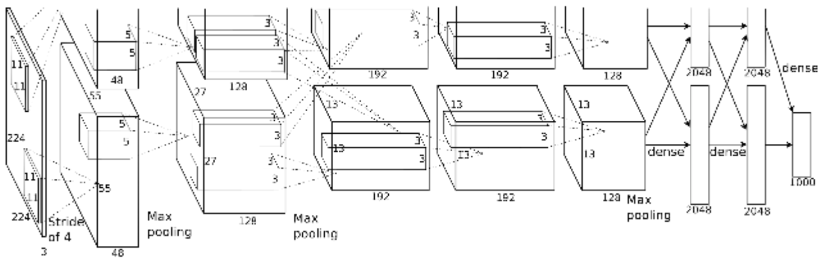


Figure 1.4: Deep neural network architecture for Alexnet. Source: Krizhevsky, Sutskever, and Hinton (2012)

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.

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**[3] . These massive models, often called "foundation models," took deep learning to new heights. GPT-3, released in 2020, contained 175 billion **parameters**[4] —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

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

[4] 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.

gigabytes in size? How do you handle the massive datasets needed for training?

The deep learning revolution of 2012 didn't emerge from nowhere—it was built 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 patterns. While it could only handle linearly separable problems—a limitation 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)**[5] .

[5] 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.

! Important 18: Convolutional Network Demo from 1989

https://www.youtube.com/watch?v=FwFduRA_L6Q&ab_channel=YannLeCun

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.

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.

1.3 The Rise of 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.

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—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?

1.4 Definition of a ML System

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 1.5: 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.

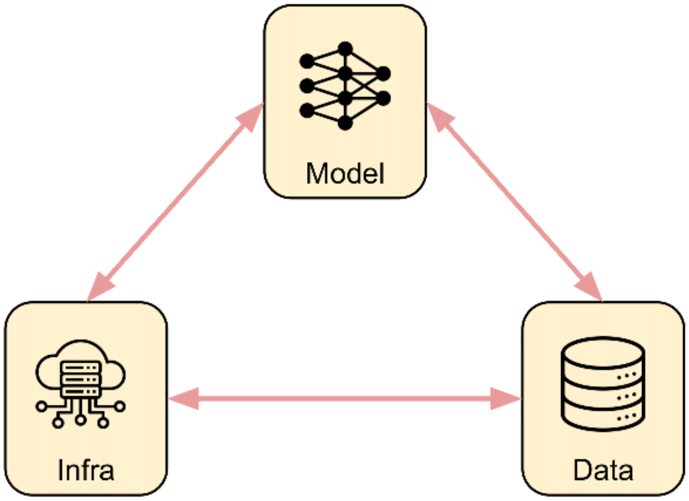


Figure 1.5: Machine learning systems involve algorithms, data, and computation, all intertwined together.

Each of these components serves a distinct but interconnected purpose:

- **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—exploring new frontiers and making discoveries. Data science teams function like mission control specialists—ensuring the constant flow of critical information and resources needed to keep the mission running. 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.

1.5 The ML Systems Lifecycle

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 1.6, 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.

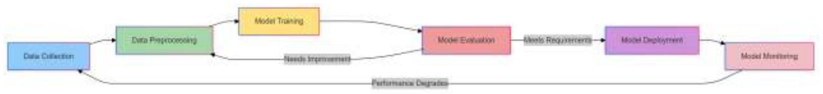


Figure 1.6: The typical lifecycle of a machine learning system.

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.

1.6 The Spectrum of ML Systems

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.

1.7 ML System Implications on the ML 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. As we discussed in our examination of technical debt, 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.

1.7.1 Emerging Trends

We are just at the beginning. As machine learning systems continue to evolve, several key trends are reshaping the landscape of ML system design and deployment.

The rise of agentic systems marks a profound evolution in ML systems. Traditional ML systems were primarily reactive—they made predictions or classifications based on input data. In contrast, agentic systems can take actions, learn from their outcomes, and adapt their behavior accordingly. These systems, exemplified by autonomous agents that can plan, reason, and execute complex tasks, introduce new architectural challenges. They require sophisticated frameworks for decision-making, safety constraints, and real-time interaction with their environment.

Architectural evolution is being driven by new hardware and deployment patterns. Specialized AI accelerators are emerging across the spectrum—from powerful data center chips to efficient edge processors to tiny neural processing units in mobile devices. This heterogeneous computing landscape is enabling new architectural possibilities, such as dynamic model distribution across tiers based on computing capabilities and current conditions. The traditional boundaries between cloud, edge, and embedded systems are becoming increasingly fluid.

Resource efficiency is gaining prominence as the environmental and economic costs of large-scale ML become more apparent. This has sparked innovation in model compression, efficient training techniques, and energy-aware computing. Future systems will likely need to balance the drive for more powerful models against growing sustainability concerns. This emphasis on efficiency is particularly relevant given our earlier discussion of technical debt and operational costs.

System intelligence is moving toward more autonomous operation. Future ML systems will likely incorporate more sophisticated self-monitoring, automated resource management, and adaptive deployment strategies. This evolution builds upon the continuous cycle we discussed earlier, but with increased automation in handling data distribution shifts, model updates, and system optimization.

Integration challenges are becoming more complex as ML systems interact with broader technology ecosystems. The need to integrate with existing software systems, handle diverse data sources, and operate across organizational boundaries is driving new approaches to system design. This integration complexity adds new dimensions to the technical debt considerations we explored earlier.

These trends suggest that future ML systems will need to be increasingly adaptable and efficient while managing growing complexity. Understanding these directions is important for building systems that can evolve with the field while avoiding the accumulation of technical debt we discussed earlier.

1.8 Real-world Applications and Impact

The ability to build and operationalize ML systems across various scales and environments has led to transformative changes across numerous sectors. This section showcases a few examples where theoretical concepts and practical considerations we have discussed manifest in tangible, impactful applications and real-world impact.

1.8.1 Case Study: FarmBeats: Edge and Embedded ML for Agriculture

FarmBeats, a project developed by Microsoft Research, shown in Figure 1.7 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.

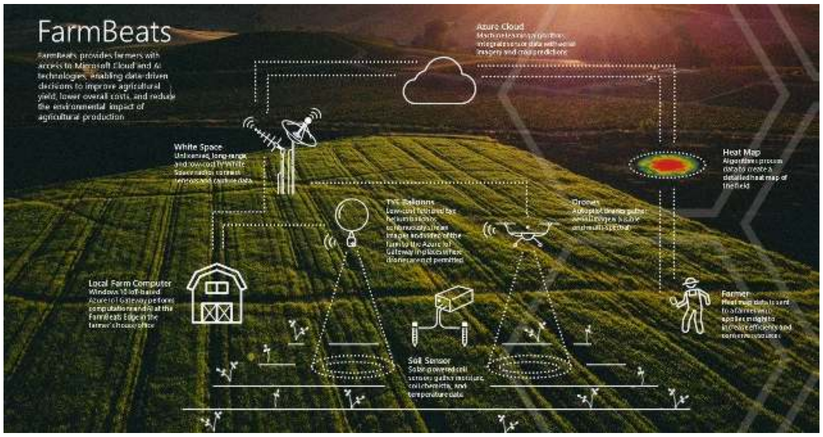

FarmBeats
FarmBeats provides farmers with access to Microsoft Cloud and AI technologies, enabling data-driven decisions to improve agricultural yield, lower overall costs, and reduce the environmental impact of agricultural production
Azure Cloud
White Space
Heat Map
Drones
Local Farm Computer
Soil Sensor
Farmer

Figure 1.7: Microsoft Farmbeats: AI, Edge & IoT for Agriculture.

Data Aspects

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 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.

Algorithm/Model Aspects

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, which may have unique soil compositions, microclimates, and farming practices.

Computing Infrastructure Aspects

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.

Impact and 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.

1.8.2 Case Study: AlphaFold: Large-Scale 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 1.8, 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.

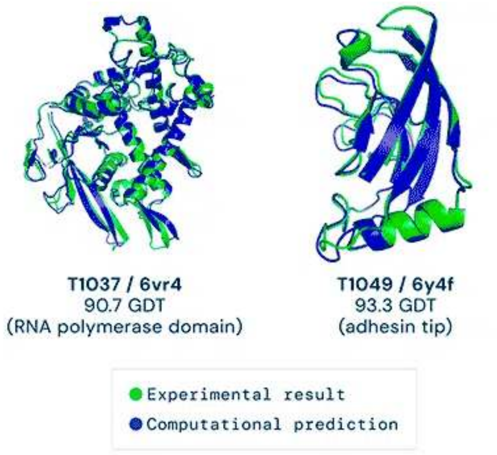


Figure 1.8: Examples of protein targets within the free modeling category. Source: Google DeepMind

Data Aspects

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.

Algorithm/Model Aspects

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.

Computing Infrastructure Aspects

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.

Impact and 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.

1.8.3 Case Study: Autonomous Vehicles: Spanning the ML Spectrum

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.

https://youtu.be/hA_-MkU0Nfw?si=6DIH7qwMbeMicnJ5

Data Aspects

The data ecosystem underpinning Waymo's technology is vast and dynamic. Each vehicle serves as a roving data center, its sensor suite—comprising 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.

Algorithm/Model Aspects

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 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.

Computing Infrastructure Aspects

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 custom AI accelerators. 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.

Impact and 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 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.

1.9 Challenges and Considerations

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.

1.9.1 Data 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 - 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," 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.

1.9.2 Model 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 (the individual settings the model learns during training). 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. This learning 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.

1.9.3 System 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.

1.9.4 Ethical and Social 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 - 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?

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.

1.10 Future Directions

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 - 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.

1.11 Learning Path and Book Structure

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

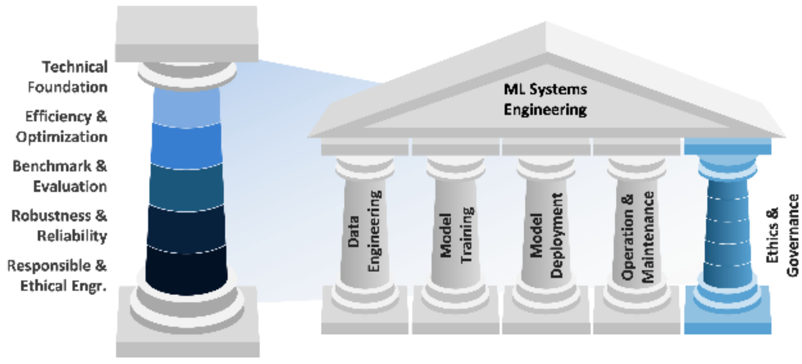


Figure 1.9: Overview of the five fundamental system pillars of Machine Learning Systems engineering.

As illustrated in Figure Figure 1.9, 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

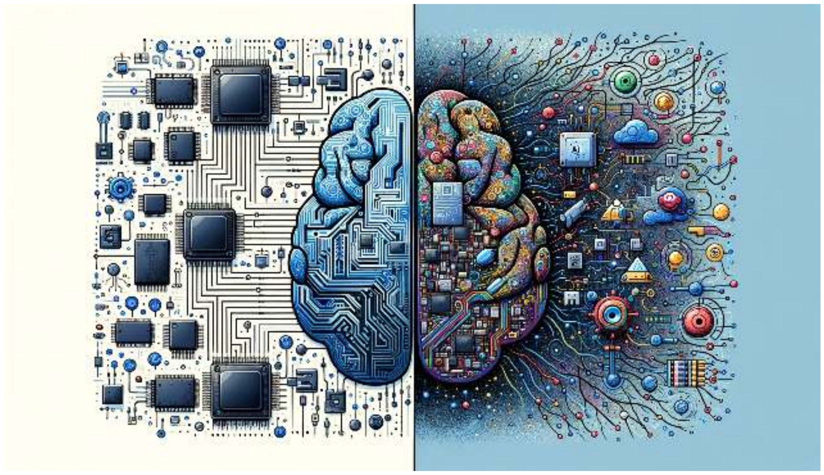


Figure 2.1: *DALL·E 3 Prompt: Illustration in a rectangular format depicting the merger of embedded systems with Embedded AI. The left half of the image portrays traditional embedded systems, including microcontrollers and processors, detailed and precise. The right half showcases the world of artificial 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.*

Machine learning (ML) systems, built on the foundation of computing systems, hold the potential to transform our world. These systems, with their specialized roles and real-time computational capabilities, represent a critical junction where data and computation meet on a micro-scale. They are specifically tailored to optimize performance, energy usage, and spatial efficiency—key factors essential for the successful implementation of ML systems.

As this chapter progresses, we will explore ML systems' complex and fascinating world. We'll gain insights into their structural design and operational features and understand their key role in powering ML applications. Starting with the basics of microcontroller units, we will examine the interfaces and peripherals that improve their functionalities. This chapter is designed to be a comprehensive guide that explains the nuanced aspects of different ML systems.

Learning Objectives

- Understand the key characteristics and differences between Cloud ML, Edge ML, and TinyML systems.
- Analyze the benefits and challenges associated with each ML paradigm.
- Explore real-world applications and use cases for Cloud ML, Edge ML, and TinyML.
- 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

ML is rapidly evolving, with new paradigms reshaping how models are developed, trained, and deployed. The field is experiencing significant innovation driven by advancements in hardware, software, and algorithmic techniques. These developments are enabling machine learning to be applied in diverse settings, from large-scale cloud infrastructures to edge devices and even tiny, resource-constrained environments.

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. At the far end, we find TinyML, which enables machine learning on extremely low-power devices with severe memory and processing constraints.

This chapter explores the landscape of contemporary machine learning systems, covering three key approaches: Cloud ML, Edge ML, and TinyML. 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.

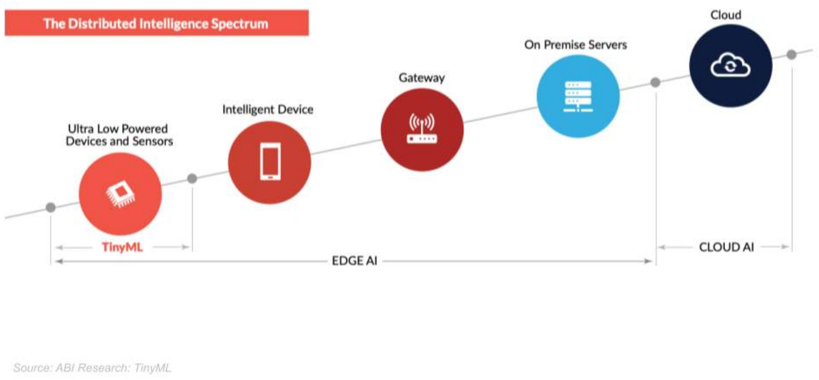

The Distributed Intelligence Spectrum
Ultra Low Powered Devices and Sensors
Intelligent Device
Gateway
On Premise Servers
Cloud
TinyML
EDGE AI
CLOUD AI
Source: ABI Research: TinyML

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

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

1. **Cloud ML:** Initially, ML was predominantly cloud-based. Powerful servers in data centers were 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 and is ideal for applications where real-time responsiveness isn't critical.

2. **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 smartphones, smart cameras, or IoT gateways. Edge ML reduces latency, 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.

3. **TinyML:** The latest development in this progression is TinyML, which enables ML models to run on extremely resource-constrained microcontrollers and small embedded systems. TinyML allows for on-device inference without relying on connectivity to the cloud or edge, opening up new possibilities for intelligent, battery-operated devices. This approach is crucial

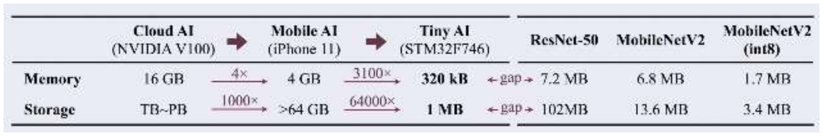
for applications where size, power consumption, and cost are critical factors.

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.
- TinyML enables AI capabilities in small, power-efficient devices, expanding the reach of ML to new domains.

The progression from Cloud to Edge to TinyML reflects a broader trend in computing towards more distributed, localized processing. This 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.

Figure 2.3 illustrates the key differences between Cloud ML, Edge ML, and TinyML in terms of hardware, latency, connectivity, power requirements, and model complexity. As we move from Cloud to Edge to TinyML, 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 TinyML. 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.



	Cloud AI (NVIDIA V100)		Mobile AI (iPhone 11)		Tiny AI (STM32F746)		ResNet-50	MobileNetV2	MobileNetV2 (int8)
Memory	16 GB	$\xrightarrow{4\times}$	4 GB	$\xrightarrow{3100\times}$	**320 kB**	←gap→	7.2 MB	6.8 MB	1.7 MB
Storage	TB~PB	$\xrightarrow{1000\times}$	>64 GB	$\xrightarrow{64000\times}$	**1 MB**	←gap→	102MB	13.6 MB	3.4 MB

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

2.2 Cloud ML

Cloud ML leverages powerful servers in the cloud for training and running large, complex ML models and relies on internet connectivity. Figure 2.4 provides an overview of Cloud ML's capabilities which we will discuss in greater detail throughout this section.

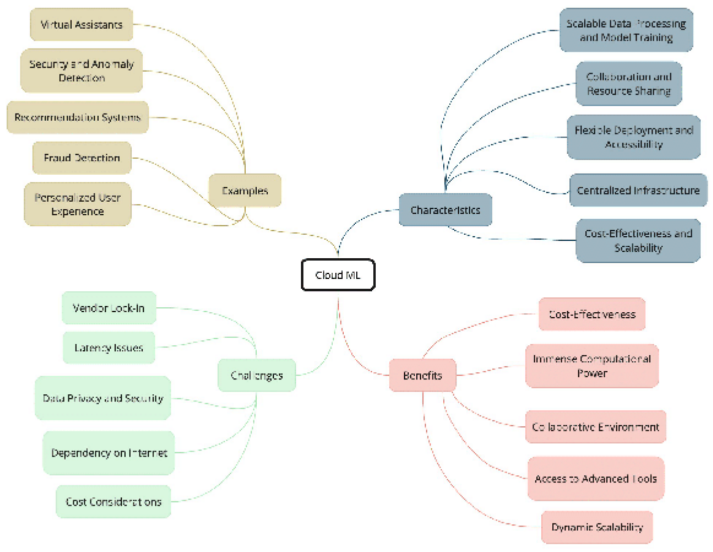

Virtual Assistants
Security and Anomaly Detection
Recommendation Systems
Fraud Detection
Personalized User Experience
Examples
Scalable Data Processing and Model Training
Collaboration and Resource Sharing
Flexible Deployment and Accessibility
Centralized Infrastructure
Cost-Effectiveness and Scalability
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

Definition of Cloud ML

Cloud Machine Learning (Cloud ML) is a subfield of machine learning that leverages the power and scalability of cloud computing infrastructure to develop, train, and deploy machine learning models. By utilizing the vast computational resources available in the cloud, Cloud ML enables the efficient handling of large-scale datasets and complex machine learning algorithms.

Centralized Infrastructure

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 data center. Cloud service providers offer a virtual platform 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.

Scalable Data Processing and Model Training

Cloud ML excels in its ability to process and analyze massive vol-

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

umes 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.

Flexible Deployment and Accessibility

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 easily deployed and made accessible to users through cloud-based services. This allows for seamless integration of machine learning capabilities into various applications and services, regardless of the user's location or device.

Collaboration and Resource Sharing

Cloud ML promotes collaboration and resource sharing among teams and organizations. The centralized nature of the cloud infrastructure enables multiple users to access and work on the same machine learning projects simultaneously. This collaborative approach facilitates knowledge sharing, accelerates the development process, and optimizes resource utilization.

Cost-Effectiveness and Scalability

By leveraging the pay-as-you-go pricing model offered by cloud service providers, Cloud ML allows organizations to avoid the upfront costs associated with building and maintaining their own machine learning infrastructure. The ability to scale resources up or down based on demand ensures cost-effectiveness and flexibility in managing machine learning projects.

Cloud ML has revolutionized the way machine learning is approached, making it more accessible, scalable, and efficient. It has opened up new possibilities for organizations to harness the power of machine learning without the need for significant investments in hardware and infrastructure.

2.2.2 Benefits

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

Immense Computational Power

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 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.

Dynamic Scalability

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.

Access to Advanced Tools and Algorithms

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 libraries, frameworks, and APIs that simplify the development and deployment of machine learning models. 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 stay at the forefront of innovation and achieve better results in their machine learning projects.

Collaborative Environment

Cloud ML fosters a collaborative environment that enables teams to work together seamlessly. The centralized nature of the cloud infrastructure allows multiple users 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, enabling efficient collaboration and driving innovation across the organization.

Cost-Effectiveness

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 hardware and infrastructure, reducing the overall cost of implementing machine learning projects. Additionally, the scalability of Cloud ML ensures that organizations can optimize their resource usage and avoid over provisioning, further enhancing cost-efficiency.

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 Issues

One of the main challenges of Cloud ML is the potential for latency issues, especially in applications that require real-time responses. Since data needs to be sent from the data source to centralized cloud servers for processing and then back to the application, there can be delays introduced by network transmission. This latency can be a significant drawback in time-sensitive scenarios, such as autonomous vehicles, real-time fraud detection, or industrial control systems, where immediate decision-making is critical. Developers need to carefully design their systems to minimize latency and ensure acceptable response times.

Data Privacy and Security Concerns

Centralizing data processing and storage in the cloud can raise concerns about data privacy and security. When sensitive data is transmitted and stored in remote data centers, it becomes vulnerable to potential cyber-attacks and unauthorized access. Cloud data centers can become attractive targets for hackers seeking to exploit vulnerabilities and gain access to valuable information. Organizations need to invest in robust security measures, such as encryption, access controls, and continuous monitoring, to protect their data in the cloud. Compliance with data privacy regulations, such as GDPR or HIPAA, also becomes a critical consideration when handling sensitive data in the cloud.

Cost Considerations

As data processing needs grow, the costs associated with using cloud services can escalate. While Cloud ML offers scalability and flexibility, organizations dealing with large data volumes may face increasing costs as they consume more cloud resources. The pay-as-you-go pricing model of cloud services means that costs can quickly add up, especially for compute-intensive tasks like model training and inference. Organizations need to carefully monitor and optimize their cloud usage to ensure cost-effectiveness. They may need to consider strategies such as data compression, efficient algorithm design, and resource allocation optimization to minimize costs while still achieving desired performance.

Dependency on Internet Connectivity

Cloud ML relies on stable and reliable internet connectivity to function effectively. Since data needs to be transmitted to and from the cloud, any disruptions or limitations in network connectivity can impact the performance and availability of the machine learning system. This dependency on internet connectivity can be a challenge in scenarios where network access is limited, unreliable, or expensive. Organizations need to ensure robust network infrastructure and consider failover mechanisms or offline capabilities to mitigate the impact of connectivity issues.

Vendor Lock-In

When adopting Cloud ML, organizations often become dependent on the specific tools, APIs, and services provided by their chosen cloud vendor. This vendor lock-in can make it difficult to switch providers or migrate to different platforms in the future. Organizations may face challenges in terms of portability, interoperability, and cost when considering a change in their cloud ML provider. It is important to carefully evaluate vendor offerings, consider long-term strategic goals, and plan for potential migration scenarios to minimize the risks associated with vendor lock-in.

Addressing these challenges requires careful planning, architectural design, and risk mitigation strategies. Organizations need to weigh the benefits of Cloud ML against the potential challenges and make informed decisions based on their specific requirements, data sensitivity, and business objectives. By proactively addressing these challenges, organizations can effectively leverage the power of Cloud ML while ensuring data privacy, security, cost-effectiveness, and overall system reliability.

2.2.4 Example 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:

Virtual Assistants

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.

Recommendation Systems

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.

Fraud Detection

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.

Personalized User Experiences

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.

Security and Anomaly Detection

Cloud ML plays a role in bolstering user security by powering anomaly detection systems. 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.

2.3 Edge ML

2.3.1 Characteristics

Definition of Edge ML

Edge Machine Learning (Edge ML) runs machine learning algorithms directly on endpoint devices or closer to where the data is generated rather than relying on centralized cloud servers. This approach brings computation closer to the data source, reducing the need to send large volumes of data over networks, often resulting in lower latency and improved data privacy. Figure 2.6 provides an overview of this section.

Decentralized Data Processing

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

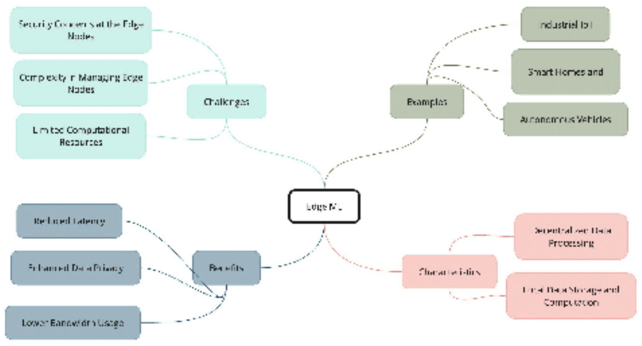

Security Concerns at the Edge Nodes
Complexity in Managing Edge Nodes
Limited Computational Resources
Challenges
Examples
Industrial IoT
Smart Homes and
Autonomous Vehicles
Edge ML
Decentralized Data Processing
Characteristics
Local Data Storage and Computation
Reduced Latency
Enhanced Data Privacy
Benefits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

Figure 2.6: Section overview for Edge ML.

central server's resources.

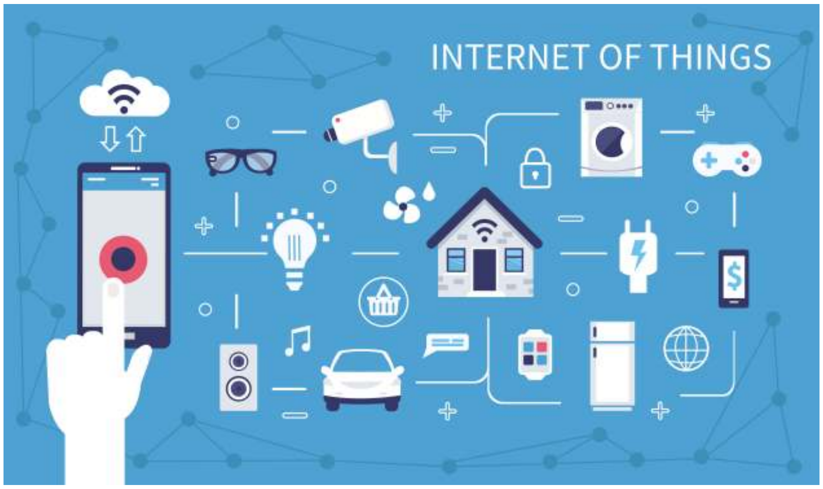


Figure 2.7: Edge ML Examples. Source: Edge Impulse.

Local Data Storage and Computation

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 often leads to more efficient computation, as data doesn't have to travel long distances, and computations are performed with a more nuanced understanding of the local context, which can sometimes result in more insightful analyses.

2.3.2 Benefits

Reduced Latency

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.

Enhanced Data Privacy

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.

Lower Bandwidth Usage

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

Limited Computational Resources Compared to Cloud ML

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.

Complexity in Managing Edge Nodes

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.

Security Concerns at the Edge Nodes

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 Example 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

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.

Smart Homes and Buildings

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.

Industrial IoT

The Industrial IoT 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 Tiny ML

2.4.1 Characteristics

Definition of TinyML

TinyML sits at the crossroads of embedded systems and machine learning, representing a burgeoning field that brings smart algorithms directly to tiny microcontrollers and sensors. These microcontrollers operate under severe resource constraints, particularly regarding memory, storage, and computational power. Figure 2.8 encapsulates the key aspects of TinyML discussed in this section.

On-Device Machine Learning

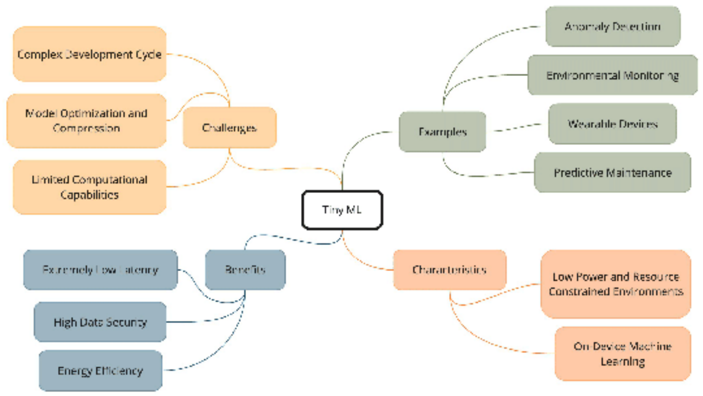


Figure 2.8: Section overview for Tiny ML.

In TinyML, the focus 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 TinyML 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.

Low Power and Resource-Constrained Environments

TinyML 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 TinyML 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. TinyML 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.

Caution 1: TinyML with Arduino

Get ready to bring machine learning to the smallest of devices! In the embedded machine learning world, TinyML is where resource constraints meet ingenuity. This Colab notebook will walk you through building a gesture recognition model designed

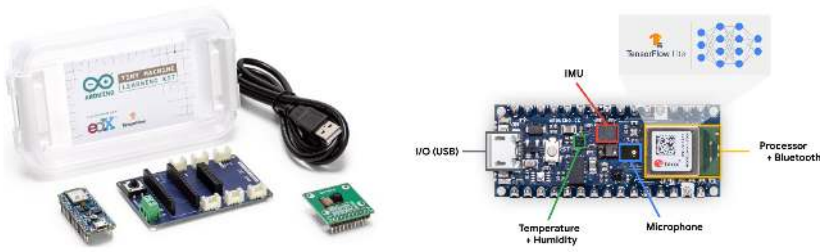


Figure 2.9: Examples of TinyML device kits. Source: Widening Access to Applied Machine Learning with TinyML.

on an Arduino board. You'll learn how to train a small but effective neural network, optimize it for minimal memory usage, and deploy it to your microcontroller. If you're excited about making everyday objects smarter, this is where it begins!

Open in Colab

2.4.2 Benefits

Extremely Low Latency

One of the standout benefits of TinyML 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.

High Data Security

TinyML 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.

Energy Efficiency

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

2.4.3 Challenges

Limited Computational Capabilities

However, the shift to TinyML 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.

Complex Development Cycle

TinyML 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.

Model Optimization and Compression

A central challenge in TinyML 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.

2.4.4 Example Use Cases

Wearable Devices

In wearables, TinyML 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, TinyML transforms how we engage with wearable tech, delivering personalized experiences directly from the device.

Predictive Maintenance

In industrial settings, TinyML plays a significant role in predictive maintenance. By deploying TinyML 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.

Anomaly Detection

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

Environmental Monitoring

In environmental monitoring, TinyML 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 TinyML, 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, TinyML 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.

2.5 Comparison

Let's bring together the different ML variants we've explored individually for a comprehensive view. Figure 2.10 illustrates the relationships and overlaps between Cloud ML, Edge ML, and TinyML using a Venn diagram. This visual representation effectively highlights the unique characteristics of each approach while also showing areas of commonality. Each ML paradigm has its own distinct features, but there are also intersections where these approaches share certain attributes or capabilities. This diagram helps us understand how these variants relate to each other in the broader landscape of machine learning implementations.

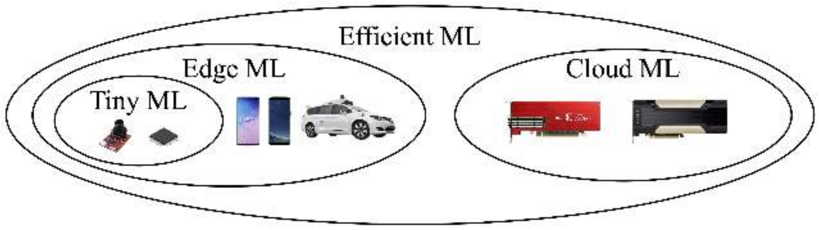


Figure 2.10: ML Venn diagram. Source: arXiv

For a more detailed comparison of these ML variants, we can refer to Table 2.1. This table offers a comprehensive analysis of Cloud ML, Edge ML, and TinyML based on various features and aspects. By examining these different characteristics side by side, we gain a clearer perspective on the unique advantages and distinguishing factors of each approach. This detailed comparison, combined with the visual overview provided by the Venn diagram, aids in making informed decisions based on the specific needs and constraints of a given application or project.

Table 2.1: Comparison of feature aspects across Cloud ML, Edge ML, and TinyML.

Aspect	Cloud ML	Edge ML	TinyML
Processing Location	Centralized servers (Data Centers)	Local devices (closer to data sources)	On-device (microcontrollers, embedded systems)
Latency	High (Depends on internet connectivity)	Moderate (Reduced latency compared to Cloud ML)	Low (Immediate processing without network delay)
Data Privacy	Moderate (Data transmitted over networks)	High (Data remains on local networks)	Very High (Data processed on-device, not transmitted)
Computational Power	High (Utilizes powerful data center infrastructure)	Moderate (Utilizes local device capabilities)	Low (Limited to the power of the embedded system)
Energy Consumption	High (Data centers consume significant energy)	Moderate (Less than data centers, more than TinyML)	Low (Highly energy-efficient, designed for low power)
Scalability	High (Easy to scale with additional server resources)	Moderate (Depends on local device capabilities)	Low (Limited by the hardware resources of the device)
Cost	High (Recurring costs for server usage, maintenance)	Variable (Depends on the complexity of local setup)	Low (Primarily upfront costs for hardware components)
Connectivity	High (Requires stable internet connectivity)	Low (Can operate with intermittent connectivity)	Very Low (Can operate without any network connectivity)
Real-time Processing	Moderate (Can be affected by network latency)	High (Capable of real-time processing locally)	Very High (Immediate processing with minimal latency)
Application Examples	Big Data Analysis, Virtual Assistants	Autonomous Vehicles, Smart Homes	Wearables, Sensor Networks

Aspect	Cloud ML	Edge ML	TinyML
Complexity	Moderate to High (Requires knowledge in cloud computing)	Moderate (Requires knowledge in local network setup)	Moderate to High (Requires expertise in embedded systems)

2.6 Conclusion

In this chapter, we've offered a panoramic view of the evolving landscape of machine learning, covering cloud, edge, and tiny ML paradigms. Cloud-based machine learning leverages the immense computational resources of cloud platforms to enable powerful and accurate models but comes with limitations, including latency and privacy concerns. Edge ML mitigates these limitations by bringing inference directly to edge devices, offering lower latency and reduced connectivity needs. TinyML takes this further by miniaturizing ML models to run directly on highly resource-constrained devices, opening up a new category of intelligent applications.

Each approach has its tradeoffs, including model complexity, latency, privacy, and hardware costs. Over time, we anticipate converging these embedded ML approaches, with cloud pre-training facilitating more sophisticated edge and tiny ML implementations. Advances like federated learning and on-device learning will enable embedded devices to refine their models by learning from real-world data.

The embedded ML landscape is rapidly evolving and poised to enable intelligent applications across a broad spectrum of devices and use cases. This chapter serves as a snapshot of the current state of embedded ML. As algorithms, hardware, and connectivity continue to improve, we can expect embedded devices of all sizes to become increasingly capable, unlocking transformative new applications for artificial intelligence.

2.7 Resources

Here is a curated list of resources to support students and instructors in their learning and teaching journeys. We are continuously working on expanding this collection and will be adding new exercises soon.

Slides

These slides are a valuable tool for instructors to deliver lectures and for students to review the material at their own pace. We encourage students and instructors to leverage these slides to improve their understanding and facilitate effective knowledge transfer.

- Embedded Systems Overview.
- Embedded Computer Hardware.
- Embedded I/O.
- Embedded systems software.
- Embedded ML software.
- Embedded Inference.
- TinyML on Microcontrollers.
- TinyML as a Service (TinyMLaaS):
  - TinyMLaaS: Introduction.
  - TinyMLaaS: 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.*

Chapter 3

DL Primer

Figure 3.1: *DALL·E 3 Prompt: Photo of a classic classroom with a large blackboard dominating one wall. Chalk drawings showcase a detailed deep neural network with several hidden layers, and each node and connection is precisely labeled with white chalk. The rustic wooden floor and brick walls provide a contrast to the modern concepts. Surrounding the room, posters mounted on frames emphasize deep learning themes: convolutional networks, transformers, neurons, activation functions, and more.*

This section serves as a primer for deep learning, providing systems practitioners with essential context and foundational knowledge needed to implement deep learning solutions effectively. Rather than delving into theoretical depths, we focus on key concepts, architectures, and practical considerations relevant to systems implementation. We begin with an overview of deep learning's evolution and its particular significance in embedded AI systems. Core concepts like neural networks are introduced with an emphasis on implementation considerations rather than mathematical foundations.

The primer explores major deep learning architectures from a systems perspective, examining their practical implications and resource requirements. We also compare deep learning to traditional machine learning approaches, helping readers make informed architectural choices based on real-world system constraints. This high-level overview sets the context for the more detailed systems-focused techniques and optimizations covered in subsequent chapters.

Learning Objectives

- Understand the basic concepts and definitions of deep neural networks.
- Recognize there are different deep learning model architectures.
- Comparison between deep learning and traditional machine learning approaches across various dimensions.
- Acquire the basic conceptual building blocks to dive deeper into advanced deep-learning techniques and applications.

3.1 Overview

3.1.1 Definition and Importance

Deep learning, a specialized area within machine learning and artificial intelligence (AI), utilizes algorithms modeled after the structure and function of the human brain, known as artificial neural networks. This field is a foundational element in AI, driving progress in diverse sectors such as computer vision, natural language processing, and self-driving vehicles. Its significance in embedded AI systems is highlighted by its capability to handle intricate calculations and predictions, optimizing the limited resources in embedded settings.

Figure 3.2 provides a visual representation of how deep learning fits within the broader context of AI and machine learning. The diagram illustrates the chronological development and relative segmentation of these three interconnected fields, showcasing deep learning as a specialized subset of machine learning, which in turn is a subset of AI.

As shown in the figure, AI represents the overarching field, encompassing all computational methods that mimic human cognitive functions. Machine learning, shown as a subset of AI, includes algorithms capable of learning from data. Deep learning, the smallest subset in the diagram, specifically involves neural networks that are able to learn more complex patterns from large volumes of data.

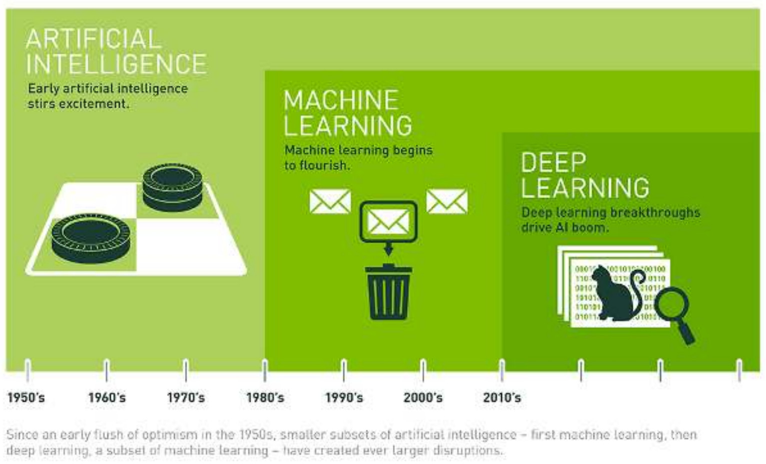

ARTIFICIAL INTELLIGENCE
Early artificial intelligence stirs excitement.
MACHINE LEARNING
Machine learning begins to flourish.
DEEP LEARNING
Deep learning breakthroughs drive AI boom.
1950's
1960's
1970's
1980's
1990's
2000's
2010's
Since an early flush of optimism in the 1950s, smaller subsets of artificial 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.

3.1.2 Brief History of Deep Learning

The idea of deep learning has origins in early artificial neural networks. It has experienced several cycles of interest, starting with the introduction of the Perceptron in the 1950s (Rosenblatt 1957), followed by the invention of backpropagation algorithms in the 1980s (Rumelhart, Hinton, and Williams 1986).

The term "deep learning" became prominent in the 2000s, characterized by advances in computational power and data accessibility. Important milestones include the successful training of deep networks like AlexNet (Krizhevsky, Sutskever, and Hinton 2012) by Geoffrey Hinton, a leading figure in AI, and the renewed focus on neural networks as effective tools for data analysis and modeling.

Deep learning has recently seen exponential growth, transforming various industries. Figure 3.3 illustrates this remarkable progression, highlighting two key trends in the field. First, the graph shows that computational growth followed an 18-month doubling pattern from 1952 to 2010. This trend then dramatically accelerated to a 6-month doubling cycle from 2010 to 2022, indicating a significant leap in computational capabilities.

Second, the figure depicts the emergence of large-scale models between 2015 and 2022. These models appeared 2 to 3 orders of magnitude faster than the general trend, following an even more aggressive 10-month doubling cycle. This rapid scaling of model sizes represents a paradigm shift in deep learning capabilities.

Multiple factors have contributed to this surge, including advance-

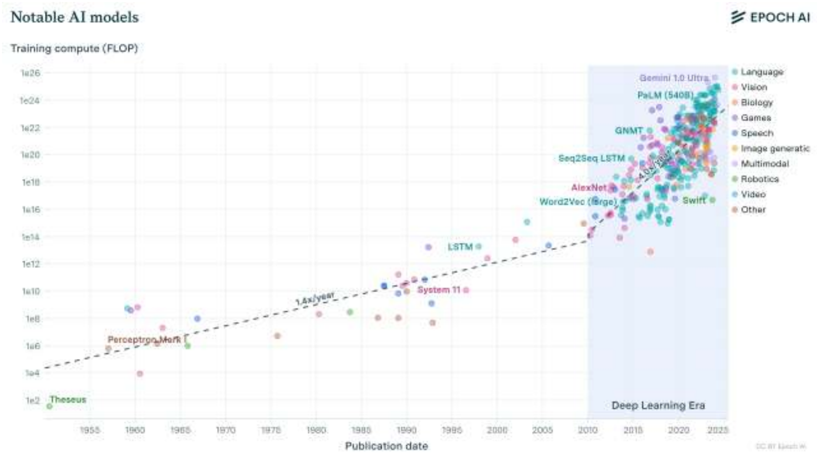


Figure 3.3: Growth of deep learning models.

ments in computational power, the abundance of big data, and improvements in algorithmic designs. First, the growth of computational capabilities, especially the arrival of Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) (N. P. Jouppi et al. 2017a), has significantly sped up the training and inference times of deep learning models. These hardware improvements have enabled the construction and training of more complex, deeper networks than what was possible in earlier years.

Second, the digital revolution has yielded a wealth of big data, offering rich material for deep learning models to learn from and excel in tasks such as image and speech recognition, language translation, and game playing. Large, labeled datasets have been key in refining and successfully deploying deep learning applications in real-world settings.

Additionally, collaborations and open-source efforts have nurtured a dynamic community of researchers and practitioners, accelerating advancements in deep learning techniques. Innovations like deep reinforcement learning, transfer learning, and generative artificial intelligence have broadened the scope of what is achievable with deep learning, opening new possibilities in various sectors, including healthcare, finance, transportation, and entertainment.

Organizations worldwide recognize deep learning's transformative potential and invest heavily in research and development to leverage its capabilities in providing innovative solutions, optimizing operations, and creating new business opportunities. As deep learning continues its upward trajectory, it is set to redefine how we interact with technology, enhancing convenience, safety, and connectivity in our lives.

3.1.3 Applications of Deep Learning

Deep learning is extensively used across numerous industries today, with its transformative impact evident in various sectors, as illustrated in Figure 3.4. In finance, it powers stock market prediction, risk assessment, and fraud detection, guiding investment strategies and improving financial decisions. Marketing leverages deep learning for customer segmentation and personalization, enabling highly targeted advertising and content optimization based on consumer behavior analysis. In manufacturing, it streamlines production processes and enhances quality control, allowing companies to boost productivity and minimize waste. Healthcare benefits from deep learning in diagnosis, treatment planning, and patient monitoring, potentially saving lives through improved medical predictions.

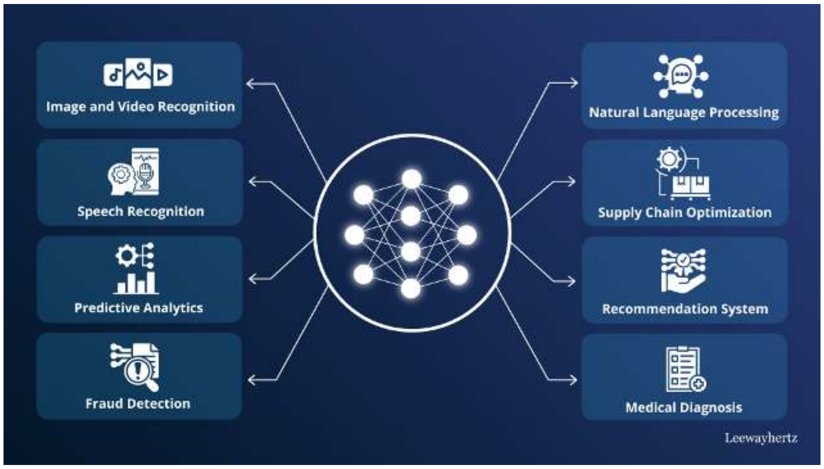


Figure 3.4: Deep learning applications, benefits, and implementations across various industries including finance, marketing, manufacturing, and healthcare. Source: Leeway Hertz

Beyond these core industries, deep learning enhances everyday products and services. Netflix uses it to strengthen its recommender systems, providing users with more personalized recommendations. Google has significantly improved its Translate service, now handling over 100 languages with increased accuracy, as highlighted in their recent advances. Autonomous vehicles from companies like Waymo, Cruise, and Motional have become a reality through deep learning in their perception system. Additionally, Amazon employs deep learning at the edge in Alexa devices for tasks such as keyword spotting. These applications demonstrate how machine learning often predicts and processes information with greater accuracy and speed than humans, revolutionizing various aspects of our daily lives.

3.1.4 Relevance to Embedded AI

Embedded AI, the integration of AI algorithms directly into hardware devices, naturally gains from deep learning capabilities. Combining deep learning algorithms and embedded systems has laid the groundwork for intelligent, autonomous devices capable of advanced on-device data processing and analysis. Deep learning aids in extracting complex patterns and information from input data, which is essential in developing smart embedded systems, from household appliances to industrial machinery. This collaboration ushers in a new era of intelligent, interconnected devices that can learn and adapt to user behavior and environmental conditions, optimizing performance and offering unprecedented convenience and efficiency.

3.2 Neural Networks

Deep learning draws inspiration from the human brain's neural networks to create decision-making patterns. This section digs into the foundational concepts of deep learning, providing insights into the more complex topics discussed later in this primer.

Neural networks serve as the foundation of deep learning, inspired by the biological neural networks in the human brain to process and analyze data hierarchically. Neural networks are composed of basic units called perceptrons, which are typically organized into layers. Each layer consists of several perceptrons, and multiple layers are stacked to form the entire network. The connections between these layers are defined by sets of weights or parameters that determine how data is processed as it flows from the input to the output of the network.

Below, we examine the primary components and structures in neural networks.

3.2.1 Perceptrons

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 3.5 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 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.

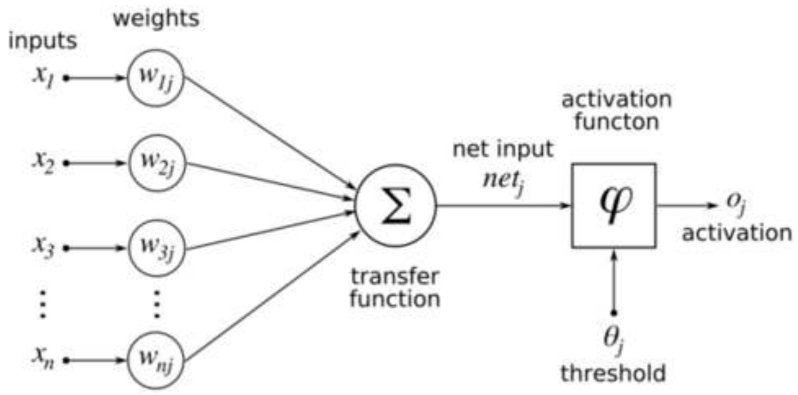


Figure 3.5: Perceptron. Conceived in the 1950s, perceptrons paved the way for developing more intricate neural networks and have been a fundamental building block in deep learning. Source: Wikimedia - Chrislb.

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:

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

This basic form of a perceptron can only model linear relationships between the input and output. Patterns found in nature are often complex and extend beyond linear relationships. To enable the perceptron to handle non-linear relationships, an activation function is applied to the linear output z.

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

Figure 3.6 illustrates an example where data exhibit a nonlinear pattern that could not be adequately modeled with a linear approach. The activation function, such as sigmoid, tanh, or ReLU, transforms the linear input sum into a non-linear output. The primary objective of this function is to introduce non-linearity into the model, enabling it to learn and perform more sophisticated tasks. Thus, the final output of the perceptron, including the activation function, can be expressed as:

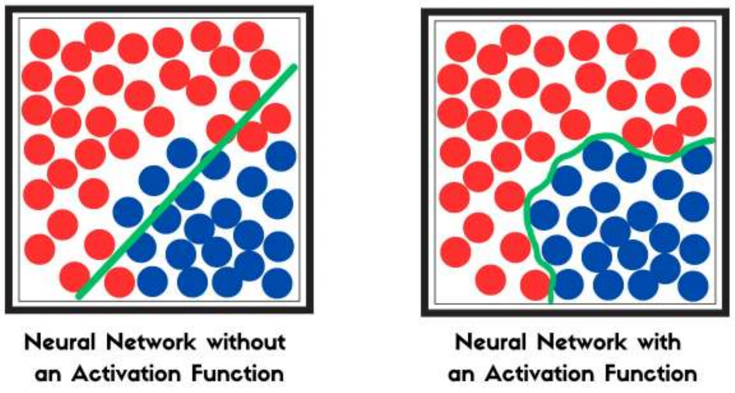

Neural Network without an Activation Function
Neural Network with an Activation Function

Figure 3.6: Activation functions enable the modeling of complex non-linear relationships. Source: Medium - Sachin Kaushik.

3.2.2 Multilayer Perceptrons

Multilayer perceptrons (MLPs) are an evolution of the single-layer perceptron model, featuring multiple layers of nodes connected in a feedforward manner. Figure 3.7 provides a visual representation of this structure. As illustrated in the figure, information in a feedforward network moves in only one direction - from the input layer on the left, through the hidden layers in the middle, to the output layer on the right, without any cycles or loops.

While a single perceptron is limited in its capacity to model complex patterns, the real strength of neural networks emerges from the assem-

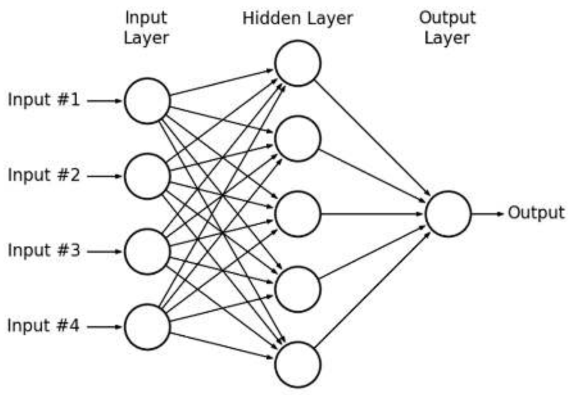


Figure 3.7: Multilayer Perceptron. Source: Wikimedia - Charlie.

bly of multiple layers. Each layer consists of numerous perceptrons working together, allowing the network to capture intricate and non-linear relationships within the data. With sufficient depth and breadth, these networks can approximate virtually any function, no matter how complex.

3.2.3 Training Process

A neural network receives an input, performs a calculation, and produces a prediction. The prediction is determined by the calculations performed within the sets of perceptrons found between the input and output layers. These calculations depend primarily on the input and the weights. Since you do not have control over the input, the objective during training is to adjust the weights in such a way that the output of the network provides the most accurate prediction.

The training process involves several key steps, beginning with the forward pass, where the existing weights of the network are used to calculate the output for a given input. This output is then compared to the true target values to calculate an error, which measures how well the network's prediction matches the expected outcome. Following this, a backward pass is performed. This involves using the error to make adjustments to the weights of the network through a process called backpropagation. This adjustment reduces the error in subsequent predictions. The cycle of forward pass, error calculation, and backward pass is repeated iteratively. This process continues until the network's predictions are sufficiently accurate or a predefined number of iterations is reached, effectively minimizing the loss function used to measure the error.

3.2.3.1 Forward Pass

The forward pass is the initial phase where data moves through the network from the input to the output layer, as illustrated in Figure 3.8. At the start of training, the network's weights are randomly initialized, setting the initial conditions for learning. During the forward pass, each layer performs specific computations on the input data using these weights and biases, and the results are then passed to the subsequent layer. The final output of this phase is the network's prediction. This prediction is compared to the actual target values present in the dataset to calculate the loss, which can be thought of as the difference between the predicted outputs and the target values. The loss quantifies the network's performance at this stage, providing a crucial metric for the subsequent adjustment of weights during the backward pass.

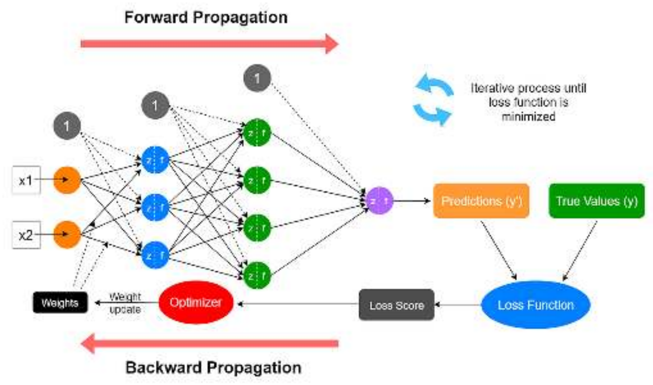


Figure 3.8: Neural networks - forward and backward propagation. Source: Linkedin

3.2.3.2 Backward Pass (Backpropagation)

After completing the forward pass and computing the loss, which measures how far the model's predictions deviate from the actual target values, the next step is to improve the model's performance by adjusting the network's weights. Since we cannot control the inputs to the model, adjusting the weights becomes our primary method for refining the model.

We determine how to adjust the weights of our model through a key algorithm called backpropagation. Backpropagation uses the calculated loss to determine the gradient of each weight. These gradients describe the direction and magnitude in which the weights should be adjusted. By tuning the weights based on these gradients, the model is better positioned to make predictions that are closer to the actual target values in the next forward pass.

Grasping these foundational concepts paves the way to understanding more intricate deep learning architectures and techniques, fostering the development of more sophisticated and productive applications, especially within embedded AI systems.

Video 2 and Video 3 build upon Video 4. They cover gradient descent and backpropagation in neural networks.

! Important 2: Gradient descent

https://www.youtube.com/watch?v=IHZwWFHWa-w&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2

! Important 3: Backpropagation

https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3

3.2.4 Model Architectures

Deep learning architectures refer to the various structured approaches that dictate how neurons and layers are organized and interact in neural networks. These architectures have evolved to tackle different problems and data types effectively. This section overviews some well-known deep learning architectures and their characteristics.

3.2.4.1 Multilayer Perceptrons (MLPs)

MLPs are basic deep learning architectures comprising three layers: an input layer, one or more hidden layers, and an output layer. These layers are fully connected, meaning each neuron in a layer is linked to every neuron in the preceding and following layers. MLPs can model intricate functions and are used in various tasks, such as regression, classification, and pattern recognition. Their capacity to learn non-linear relationships through backpropagation makes them a versatile instrument in the deep learning toolkit.

In embedded AI systems, MLPs can function as compact models for simpler tasks like sensor data analysis or basic pattern recognition, where computational resources are limited. Their ability to learn non-linear relationships with relatively less complexity makes them a suitable choice for embedded systems.

Caution 2: Multilayer Perceptrons (MLPs)

We've just scratched the surface of neural networks. Now, you'll get to try and apply these concepts in practical examples. In the provided Colab notebooks, you'll explore:

Predicting house prices: Learn how neural networks can analyze housing data to estimate property values. Open in Colab

Image Classification: Discover how to build a network to understand the famous MNIST handwritten digit dataset. Open in Colab

Real-world medical diagnosis: Use deep learning to tackle the important task of breast cancer classification. Open in Colab

3.2.4.2 Convolutional Neural Networks (CNNs)

CNNs are mainly used in image and video recognition tasks. This architecture consists of two main parts: the convolutional base and the fully connected layers. In the convolutional base, convolutional layers filter input data to identify features like edges, corners, and textures. Following each convolutional layer, a pooling layer can be applied to reduce the spatial dimensions of the data, thereby decreasing computational load and concentrating the extracted features. Unlike MLPs, which treat input features as flat, independent entities, CNNs maintain the spatial relationships between pixels, making them particularly effective for image and video data. The extracted features from the convolutional base are then passed into the fully connected layers, similar to those used in MLPs, which perform classification based on the features extracted by the convolution layers. CNNs have proven highly effective in image recognition, object detection, and other computer vision applications.

Video 4 explains how neural networks work using handwritten digit recognition as an example application. It also touches on the math underlying neural nets.

Important 4: MLP & CNN Networks

https://www.youtube.com/embed/aircAruvnKk?si=ZRj8jf4yx7ZMe8EK

CNNs are crucial for image and video recognition tasks, where real-time processing is often needed. They can be optimized for embedded systems using techniques like quantization and pruning to minimize memory usage and computational demands, enabling efficient object detection and facial recognition functionalities in devices with limited computational resources.

Caution 3: Convolutional Neural Networks (CNNs)

We discussed that CNNs excel at identifying image features, making them ideal for tasks like object classification. Now, you'll get to put this knowledge into action! This Colab notebook focuses on building a CNN to classify images from the CIFAR-10 dataset, which includes objects like airplanes, cars, and animals. You'll learn about the key differences between CIFAR-10 and the MNIST dataset we explored earlier and how these differences influence model choice. By the end of this notebook, you'll have a grasp of CNNs for image recognition.

Open in Colab

3.2.4.3 Recurrent Neural Networks (RNNs)

RNNs are suitable for sequential data analysis, like time series forecasting and natural language processing. In this architecture, connections between nodes form a directed graph along a temporal sequence, allowing information to be carried across sequences through hidden state vectors. Variants of RNNs include Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU), designed to capture longer dependencies in sequence data.

These networks can be used in voice recognition systems, predictive maintenance, or IoT devices where sequential data patterns are common. Optimizations specific to embedded platforms can assist in managing their typically high computational and memory requirements.

3.2.4.4 Generative Adversarial Networks (GANs)

GANs consist of two networks, a generator and a discriminator, trained simultaneously through adversarial training (Goodfellow et al. 2020). The generator produces data that tries to mimic the real data distribution, while the discriminator distinguishes between real and generated data. GANs are widely used in image generation, style transfer, and data augmentation.

In embedded settings, GANs could be used for on-device data augmentation to improve the training of models directly on the embedded device, enabling continual learning and adaptation to new data without the need for cloud computing resources.

3.2.4.5 Autoencoders

Autoencoders are neural networks for data compression and noise reduction (Bank, Koenigstein, and Giryes 2023). They are structured to encode input data into a lower-dimensional representation and then decode it back to its original form. Variants like Variational Autoencoders (VAEs) introduce probabilistic layers that allow for generative properties, finding applications in image generation and anomaly detection.

Using autoencoders can help in efficient data transmission and storage, improving the overall performance of embedded systems with limited computational and memory resources.

3.2.4.6 Transformer Networks

Transformer networks have emerged as a powerful architecture, especially in natural language processing (Vaswani et al. 2017). These networks use self-attention mechanisms to weigh the influence of different input words on each output word, enabling parallel computation and capturing intricate patterns in data. Transformer networks have led to state-of-the-art results in tasks like language translation, summarization, and text generation.

These networks can be optimized to perform language-related tasks directly on the device. For example, transformers can be used in embedded systems for real-time translation services or voice-assisted interfaces, where latency and computational efficiency are crucial. Techniques such as model distillation can be employed to deploy these networks on embedded devices with limited resources.

These architectures serve specific purposes and excel in different domains, offering a rich toolkit for addressing diverse problems in embedded AI systems. Understanding the nuances of these architectures is crucial in designing effective and efficient deep learning models for various applications.

3.2.5 Traditional ML vs Deep Learning

Deep learning extends traditional machine learning by utilizing neural networks to discern patterns in data. In contrast, traditional machine learning relies on a set of established algorithms such as decision trees, k-nearest neighbors, and support vector machines, but does not involve neural networks. Figure 3.9 provides a visual comparison of Machine Learning and Deep Learning, highlighting their key differences in approach and capabilities.

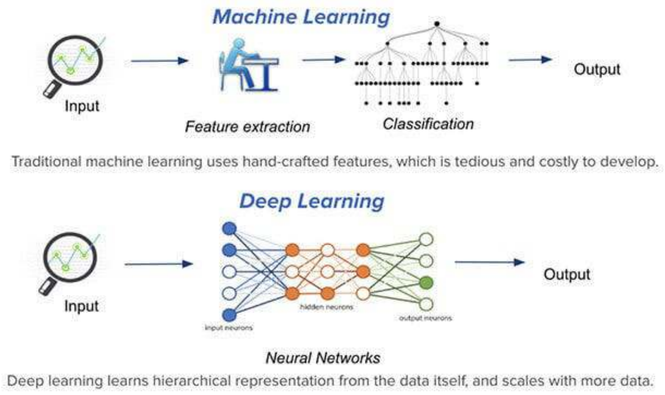

Machine Learning
Input
Feature extraction
Classification
Output
Traditional machine learning uses hand-crafted features, which is tedious and costly to develop.
Deep Learning
Input
input neurons
hidden neurons
output neurons
Output
Neural Networks
Deep learning learns hierarchical representation from the data itself, and scales with more data.

Figure 3.9: Comparing Machine Learning and Deep Learning. Source: Medium

As shown in the figure, deep learning models can process raw data directly and automatically extract relevant features, while traditional machine learning often requires manual feature engineering. The figure also illustrates how deep learning models can handle more complex tasks and larger datasets compared to traditional machine learning approaches.

To further highlight the differences, Table 3.1 provides a more detailed comparison of the contrasting characteristics between traditional ML and deep learning. This table complements the visual representation in Figure 3.9 by offering specific points of comparison across various aspects of these two approaches.

Table 3.1: Comparison of traditional machine learning and deep learning.

Aspect	Traditional ML	Deep Learning
Data Requirements	Low to Moderate (efficient with smaller datasets)	High (requires large datasets for nuanced learning)
Model Complexity	Moderate (suitable for well-defined problems)	High (detects intricate patterns, suited for complex tasks)
Computational Resources	Low to Moderate (cost-effective, less resource-intensive)	High (demands substantial computational power and resources)
Deployment Speed	Fast (quicker training and deployment cycles)	Slow (prolonged training times, esp. with larger datasets)
Interpretability	High (clear insights into decision pathways)	Low (complex layered structures, "black box" nature)
Maintenance	Easier (simple to update and maintain)	Complex (requires more efforts in maintenance and updates)

3.2.6 Choosing Traditional ML vs. DL

3.2.6.1 Data Availability and Volume

Amount of Data: Traditional machine learning algorithms, such as decision trees or Naive Bayes, are often more suitable when data availability is limited. They offer robust predictions even with smaller datasets. This is particularly true in medical diagnostics for disease prediction and customer segmentation in marketing.

Data Diversity and Quality: Traditional machine learning algorithms often work well with structured data (the input to the model is a set of features, ideally independent of each other) but may require significant preprocessing effort (i.e., feature engineering). On the other hand, deep learning takes the approach of automatically performing feature engineering as part of the model architecture. This approach enables the construction of end-to-end models capable of directly mapping from unstructured input data (such as text, audio, and images) to the desired output without relying on simplistic heuristics that have limited effectiveness. However, this results in larger models demanding more data and computational resources. In noisy data, the necessity for larger datasets is further emphasized when utilizing Deep Learning.

3.2.6.2 Complexity of the Problem

Problem Granularity: Problems that are simple to moderately complex, which may involve linear or polynomial relationships between variables, often find a better fit with traditional machine learning methods.

Hierarchical Feature Representation: Deep learning models are excellent in tasks that require hierarchical feature representation, such as image and speech recognition. However, not all problems require this complexity, and traditional machine learning algorithms may sometimes offer simpler and equally effective solutions.

3.2.6.3 Hardware and Computational Resources

Resource Constraints: The availability of computational resources often influences the choice between traditional ML and deep learning. The former is generally less resource-intensive and thus preferable in environments with hardware limitations or budget constraints.

Scalability and Speed: Traditional machine learning algorithms, like support vector machines (SVM), often allow for faster training times and easier scalability, which is particularly beneficial in projects with tight timelines and growing data volumes.

3.2.6.4 Regulatory Compliance

Regulatory compliance is crucial in various industries, requiring adherence to guidelines and best practices such as the General Data Protection Regulation (GDPR) in the EU. Traditional ML models, due to their inherent interpretability, often align better with these regulations, especially in sectors like finance and healthcare.

3.2.6.5 Interpretability

Understanding the decision-making process is easier with traditional machine learning techniques than deep learning models, which function as “black boxes,” making it challenging to trace decision pathways.

3.2.7 Making an Informed Choice

Given the constraints of embedded AI systems, understanding the differences between traditional ML techniques and deep learning becomes essential. Both avenues offer unique advantages, and their distinct characteristics often dictate the choice of one over the other in different scenarios.

Despite this, deep learning has steadily outperformed traditional machine learning methods in several key areas due to abundant data, computational advancements, and proven effectiveness in complex tasks. Here are some specific reasons why we focus on deep learning:

1. **Superior Performance in Complex Tasks:** Deep learning models, particularly deep neural networks, excel in tasks where the relationships between data points are incredibly intricate. Tasks like image and speech recognition, language translation, and playing complex games like Go and Chess have seen significant advancements primarily through deep learning algorithms.

2. **Efficient Handling of Unstructured Data:** Unlike traditional machine learning methods, deep learning can more effectively process unstructured data. This is crucial in today's data landscape, where the vast majority of data, such as text, images, and videos, is unstructured.

3. **Leveraging Big Data:** With the availability of big data, deep learning models can learn and improve continually. These models excel at utilizing large datasets to improve their predictive accuracy, a limitation in traditional machine-learning approaches.

4. **Hardware Advancements and Parallel Computing:** The advent of powerful GPUs and the availability of cloud computing platforms have enabled the rapid training of deep learning models. These advancements have addressed one of deep learning's significant challenges: the need for substantial computational resources.

5. **Dynamic Adaptability and Continuous Learning:** Deep learning models can dynamically adapt to new information or data. They can be trained to generalize their learning to new, unseen data, crucial in rapidly evolving fields like autonomous driving or real-time language translation.

While deep learning has gained significant traction, it's essential to understand that traditional machine learning is still relevant. As we dive deeper into the intricacies of deep learning, we will also highlight situations where traditional machine learning methods may be more appropriate due to their simplicity, efficiency, and interpretability. By focusing on deep learning in this text, we aim to equip readers with the knowledge and tools to tackle modern, complex problems across various domains while also providing insights into the comparative advantages and appropriate application scenarios for deep learning and traditional machine learning techniques.

3.3 Conclusion

Deep learning has become a potent set of techniques for addressing intricate pattern recognition and prediction challenges. Starting with an overview, we outlined the fundamental concepts and principles governing deep learning, laying the groundwork for more advanced studies.

Central to deep learning, we explored the basic ideas of neural networks, powerful computational models inspired by the human brain's interconnected neuron structure. This exploration allowed us to appreciate neural networks' capabilities and potential in creating sophisticated algorithms capable of learning and adapting from data.

Understanding the role of libraries and frameworks was a key part of our discussion. We offered insights into the tools that can facilitate developing and deploying deep learning models. These resources ease the implementation of neural networks and open avenues for innovation and optimization.

Next, we tackled the challenges one might face when embedding deep learning algorithms within embedded systems, providing a critical perspective on the complexities and considerations of bringing AI to edge devices.

Furthermore, we examined deep learning's limitations. Through discussions, we unraveled the challenges faced in deep learning applications and outlined scenarios where traditional machine learning might outperform deep learning. These sections are crucial for fostering a balanced view of deep learning's capabilities and limitations.

In this primer, we have equipped you with the knowledge to make informed choices between deploying traditional machine learning or deep learning techniques, depending on the unique demands and constraints of a specific problem.

As we conclude this chapter, we hope you are now well-equipped with the basic "language" of deep learning and prepared to go deeper into the subsequent chapters with a solid understanding and critical perspective. The journey ahead is filled with exciting opportunities and challenges in embedding AI within systems.

3.4 Resources

Here is a curated list of resources to support students and instructors in their learning and teaching journeys. We are continuously working

on expanding this collection and will be adding new exercises soon.

Slides

These slides are a valuable tool for instructors to deliver lectures and for students to review the material at their own pace. We encourage students and instructors to leverage these slides to improve their understanding and facilitate effective knowledge transfer.

- 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 4
- Video 2
- Video 3

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.

- Exercise 2
- Exercise 3

Chapter 4

AI Workflow

Figure 4.1: *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.*

The ML workflow is a structured approach that guides professionals and researchers through developing, deploying, and maintaining ML models. This workflow is generally divided into several crucial stages, each contributing to the effective development of intelligent systems.

In this chapter, we will explore the machine learning workflow, setting the stage for subsequent chapters that go deeper into the specifics. This chapter focuses only presenting a high-level overview of the steps involved in the ML workflow.

Learning Objectives

- Understand the ML workflow and gain insights into the structured approach and stages of developing, deploying, and maintaining machine learning models.
- Learn about the unique challenges and distinctions between workflows for Traditional machine learning and embedded AI.
- Appreciate the roles in ML projects and understand their responsibilities and significance.
- Understanding the importance, applications, and considerations for implementing ML models in resource-constrained environments.
- Gain awareness about the ethical and legal aspects that must be considered and adhered to in ML and embedded AI projects.
- Establish a basic understanding of ML workflows and roles to be well-prepared for deeper exploration in the following chapters.

4.1 Overview

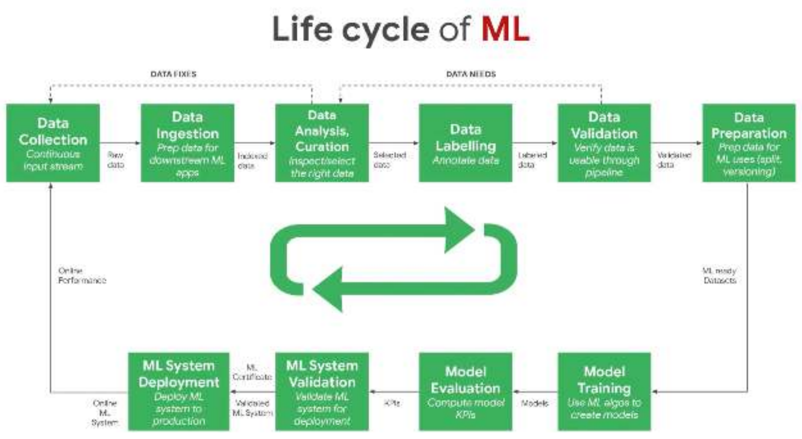


Figure 4.2: Multi-step design methodology for the development of a machine learning model. Commonly referred to as the machine learning lifecycle

Figure 4.2 illustrates the systematic workflow required for developing a successful machine learning model. This end-to-end process, commonly referred to as the machine learning lifecycle, enables you to build, deploy, and maintain models effectively. It typically involves the following key steps:

1. **Problem Definition** - Start by clearly articulating the specific problem you want to solve. This focuses on your efforts during data collection and model building.
2. **Data Collection and Preparation:** Gather relevant, high-quality training data that captures all aspects of the problem. Clean and preprocess the data to prepare it for modeling.
3. **Model Selection and Training:** Choose a machine learning algorithm suited to your problem type and data. Consider the pros and cons of different approaches. Feed the prepared data into the model to train it. Training time varies based on data size and model complexity.
4. **Model Evaluation:** Test the trained model on new unseen data to measure its predictive accuracy. Identify any limitations.
5. **Model Deployment:** Integrate the validated model into applications or systems to start operationalization.
6. **Monitor and Maintain:** Track model performance in production. Retrain periodically on new data to keep it current.

Following this structured ML workflow helps guide you through the key phases of development. It ensures you build effective and robust models ready for real-world deployment, resulting in higher-quality models that solve your business needs.

The ML workflow is iterative, requiring ongoing monitoring and potential adjustments. Additional considerations include:

- **Version Control:** Track code and data changes to reproduce results and revert to earlier versions if needed.
- **Documentation:** Maintain detailed documentation for workflow understanding and reproduction.
- **Testing:** Rigorously test the workflow to ensure its functionality.
- **Security:** Safeguard your workflow and data when deploying models in production settings.

4.2 Traditional vs. Embedded AI

The ML workflow is a universal guide applicable across various platforms, including cloud-based solutions, edge computing, and TinyML. However, the workflow for Embedded AI introduces unique complexities and challenges, making it a captivating domain and paving the way for remarkable innovations. Figure 4.3 illustrates the differences between Machine Learning and Deep Learning.

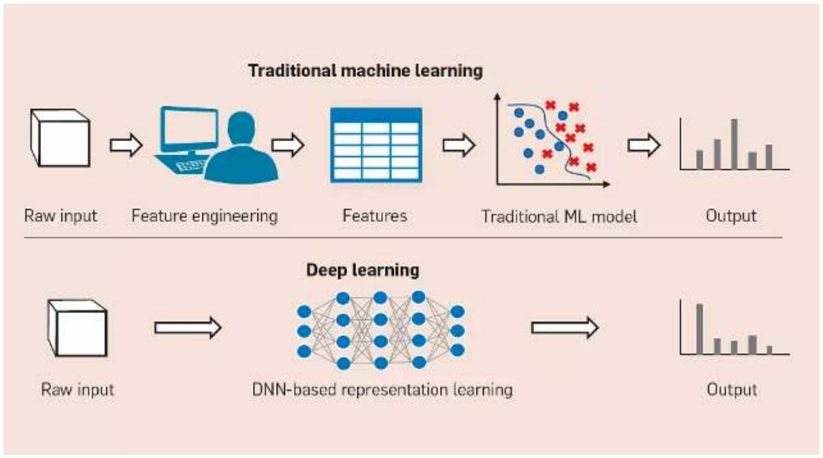


Figure 4.3: Comparing traditional Machine Learning and Deep Learning. Source: BBN Times

Figure 4.4 showcases the uses of embedded ai in various industries.

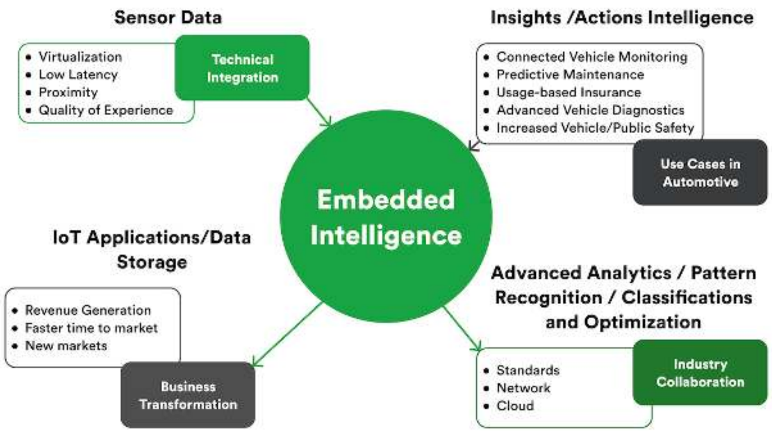


Figure 4.4: Embedded AI applications. Source: Rinf.tech

4.2.1 Resource Optimization

- **Traditional ML Workflow:** This workflow prioritizes model accuracy and performance, often leveraging abundant computational resources in cloud or data center environments.
- **Embedded AI Workflow:** Given embedded systems' resource constraints, this workflow requires careful planning to optimize

model size and computational demands. Techniques like model quantization and pruning are crucial.

4.2.2 Real-time Processing

- **Traditional ML Workflow:** Less emphasis on real-time processing, often relying on batch data processing.
- **Embedded AI Workflow:** Prioritizes real-time data processing, making low latency and quick execution essential, especially in applications like autonomous vehicles and industrial automation.

4.2.3 Data Management and Privacy

- **Traditional ML Workflow:** Processes data in centralized locations, often necessitating extensive data transfer and focusing on data security during transit and storage.
- **Embedded AI Workflow:** This workflow leverages edge computing to process data closer to its source, reducing data transmission and enhancing privacy through data localization.

4.2.4 Hardware-Software Integration

- **Traditional ML Workflow:** Typically operates on general-purpose hardware, with software development occurring independently.
- **Embedded AI Workflow:** This workflow involves a more integrated approach to hardware and software development, often incorporating custom chips or hardware accelerators to achieve optimal performance.

4.3 Roles & Responsibilities

Creating an ML solution, especially for embedded AI, is a multidisciplinary effort involving various specialists. Unlike traditional software development, building an ML solution demands a multidisciplinary approach due to the experimental nature of model development and the resource-intensive requirements of training and deploying these models.

There is a pronounced need for roles focusing on data for the success of machine learning pipelines. Data scientists and data engineers handle data collection, build data pipelines, and ensure data quality. Since the nature of machine learning models depends on the data they consume, the models are unique and vary with different applications, necessitating extensive experimentation. Machine learning researchers and engineers drive this experimental phase through continuous testing, validation, and iteration to achieve optimal performance.

The deployment phase often requires specialized hardware and infrastructure, as machine learning models can be resource-intensive, demanding high computational power and efficient resource management. This necessitates collaboration with hardware engineers to ensure that the infrastructure can support the computational demands of model training and inference.

As models make decisions that can impact individuals and society, ethical and legal aspects of machine learning are becoming increasingly important. Ethicists and legal advisors are needed to ensure compliance with ethical standards and legal regulations.

Understanding the various roles involved in an ML project is crucial for its successful completion. Table 4.1 provides a general overview of these typical roles, although it's important to note that the lines between them can sometimes blur. Let's examine this breakdown:

Table 4.1: Roles and responsibilities of people involved in MLOps.

Role	Responsibilities
Project Manager	Oversees the project, ensuring timelines and milestones are met.
Domain Experts	Offer domain-specific insights to define project requirements.
Data Scientists	Specialize in data analysis and model development.
Machine Learning Engineers	Focus on model development and deployment.
Data Engineers	Manage data pipelines.
Embedded Systems Engineers	Integrate ML models into embedded systems.
Software Developers	Develop software components for AI system integration.
Hardware Engineers	Design and optimize hardware for the embedded AI system.
UI/UX Designers	Focus on user-centric design.
QA Engineers	Ensure the system meets quality standards.
Ethicists and Legal Advisors	Consult on ethical and legal compliance.

Role	Responsibilities
Operations and Maintenance Personnel	Monitor and maintain the deployed system.
Security Specialists	Ensure system security.

This holistic view facilitates seamless collaboration and nurtures an environment ripe for innovation and breakthroughs. As we proceed through the upcoming chapters, we will explore each role's essence and expertise and foster a deeper understanding of the complexities involved in AI projects. For a more detailed discussion of the specific tools and techniques these roles use, as well as an in-depth exploration of their responsibilities, refer to Section 13.5.

4.4 Conclusion

This chapter has laid the foundation for understanding the machine learning workflow, a structured approach crucial for the development, deployment, and maintenance of ML models. We explored the unique challenges faced in ML workflows, where resource optimization, real-time processing, data management, and hardware-software integration are paramount. These distinctions underscore the importance of tailoring workflows to meet the specific demands of the application environment.

Moreover, we emphasized the significance of multidisciplinary collaboration in ML projects. By examining the diverse roles involved, from data scientists to software engineers, we gained an overview of the teamwork necessary to navigate the experimental and resource-intensive nature of ML development. This understanding is crucial for fostering effective communication and collaboration across different domains of expertise.

As we move forward to more detailed discussions in subsequent chapters, this high-level overview equips us with a holistic perspective on the ML workflow and the various roles involved. This foundation will prove important as we dive into specific aspects of machine learning, which will allow us to contextualize advanced concepts within the broader framework of ML development and deployment.

4.5 Resources

Here is a curated list of resources to support students and instructors in their learning and teaching journeys. We are continuously working

on expanding this collection and will add new exercises soon.

Slides

These slides are a valuable tool for instructors to deliver lectures and for students to review the material at their own pace. We encourage students and instructors to leverage these slides to improve their understanding and facilitate effective knowledge transfer.

- ML Workflow.
- ML Lifecycle.

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.*

Chapter 5

Data Engineering

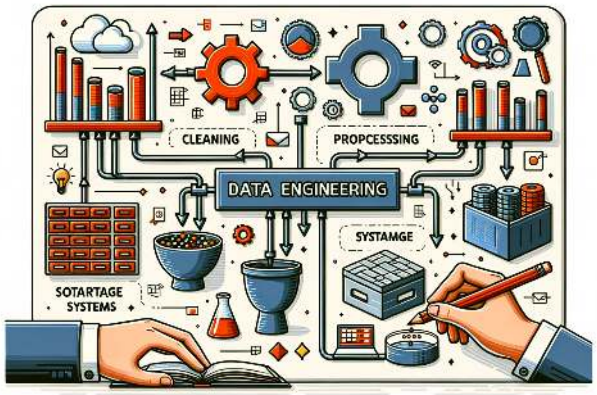


Figure 5.1: *DALL·E 3 Prompt: Create a rectangular illustration visualizing the concept of data engineering. Include elements such as raw data sources, data processing pipelines, storage systems, and refined datasets. Show how raw data is transformed through cleaning, processing, and storage to become valuable information that can be analyzed and used for decision-making.*

Data is the lifeblood of AI systems. Without good data, even the most advanced machine-learning algorithms will not succeed. However, TinyML models operate on devices with limited processing power and memory. This section explores the intricacies of building high-quality datasets to fuel our AI models. Data engineering involves collecting, storing, processing, and managing data to train machine learning models.

Learning Objectives

- Understand the importance of clearly defining the problem statement and objectives when embarking on an ML project.
- Recognize various data sourcing techniques, such as web scraping, crowdsourcing, and synthetic data generation, along with their advantages and limitations.
- Appreciate the need for thoughtful data labeling, using manual or AI-assisted approaches, to create high-quality training datasets.
- Briefly learn different methods for storing and managing data, such as databases, data warehouses, and data lakes.
- Comprehend the role of transparency through metadata and dataset documentation and tracking data provenance to facilitate ethics, auditing, and reproducibility.
- Understand how licensing protocols govern legal data access and usage, necessitating careful compliance.
- Recognize key challenges in data engineering, including privacy risks, representation gaps, legal restrictions around data access, and balancing competing priorities.

5.1 Overview

Imagine a world where AI can diagnose diseases with unprecedented accuracy, but only if the data used to train it is unbiased and reliable. This is where data engineering comes in. While over 90% of the world's data has been created in the past two decades, this vast amount of information is only helpful for building effective AI models with proper processing and preparation. Data engineering bridges this gap by transforming raw data into a high-quality format that fuels AI innovation. In today's data-driven world, protecting user privacy is paramount. Whether mandated by law or driven by user concerns, anonymization techniques like differential privacy and aggregation are vital in mitigating privacy risks. However, careful implementation is crucial to ensure these methods don't compromise data utility. Dataset creators face complex privacy and representation challenges when building high-quality training data, especially for sensitive domains like healthcare.

Legally, creators may need to remove direct identifiers like names and ages. Even without legal obligations, removing such information can help build user trust. However, excessive anonymization can compromise dataset utility. Techniques like differential privacy[1], aggregation, and reducing detail provide alternatives to balance privacy and utility but have downsides. Creators must strike a thoughtful balance based on the use case.

While privacy is paramount, ensuring fair and robust AI models requires addressing representation gaps in the data. It is crucial yet insufficient to ensure diversity across individual variables like gender, race, and accent. These combinations, sometimes called higher-order gaps, can significantly impact model performance. For example, a medical dataset could have balanced gender, age, and diagnosis data individually, but it lacks enough cases to capture older women with a specific condition. Such higher-order gaps are not immediately obvious but can critically impact model performance.

Creating useful, ethical training data requires holistic consideration of privacy risks and representation gaps. Elusive perfect solutions necessitate conscientious data engineering practices like anonymization, aggregation, under-sampling of overrepresented groups, and synthesized data generation to balance competing needs. This facilitates models that are both accurate and socially responsible. Cross-functional collaboration and external audits can also strengthen training data. The challenges are multifaceted but surmountable with thoughtful effort.

We begin by discussing data collection: Where do we source data, and how do we gather it? Options range from scraping the web, accessing APIs, and utilizing sensors and IoT devices to conducting surveys and gathering user input. These methods reflect real-world practices. Next, we dive into data labeling, including considerations for human involvement. We'll discuss the trade-offs and limitations of human labeling and explore emerging methods for automated labeling. Following that, we'll address data cleaning and preprocessing, a crucial yet frequently undervalued step in preparing raw data for AI model training. Data augmentation comes next, a strategy for enhancing limited datasets by generating synthetic samples. This is particularly pertinent for embedded systems, as many use cases need extensive data repositories readily available for curation. Synthetic data generation emerges as a viable alternative with advantages and disadvantages. We'll also touch upon dataset versioning, emphasizing the importance of tracking data modifications over time. Data is ever-evolving; hence, it's imperative to devise strategies for managing and storing expansive datasets. By the end of this section, you'll possess a comprehensive understanding of the entire data pipeline, from collection to storage, essential for operationalizing AI systems. Let's embark on this journey!

5.2 Problem Definition

In many machine learning domains, sophisticated algorithms take center stage, while the fundamental importance of data quality is often overlooked. This neglect gives rise to "Data Cascades" by Sambasivan et al. (2021a)—events where lapses in data quality compound, leading to negative downstream consequences such as flawed predictions, project terminations, and even potential harm to communities.

Figure 5.2 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 depicted 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 the cascading effects illustrated in the figure.

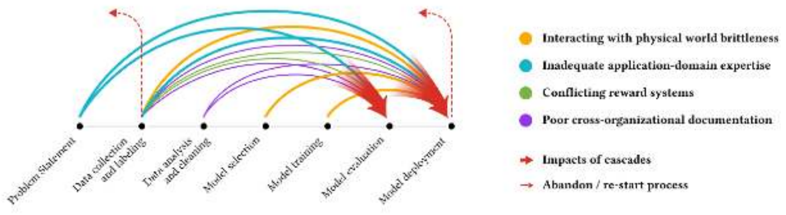

Problem Statement
Data collection and labeling
Data analysis and cleaning
Model selection
Model training
Model evaluation
Model deployment
Interacting with physical world brittleness
Inadequate application-domain expertise
Conflicting reward systems
Poor cross-organizational documentation
Impacts of cascades
Abandon / re-start process

Figure 5.2: Data cascades: compounded costs. Source: Sambasivan et al. (2021a).

Despite many ML professionals recognizing the importance of data, numerous practitioners report facing these cascades. This highlights a systemic issue: while the allure of developing advanced models remains, data often needs to be more appreciated.

Keyword Spotting (KWS) provides an excellent example of TinyML in action, as illustrated in Figure 5.3. This technology is critical for voice-enabled interfaces on endpoint devices such as smartphones. Typically functioning as lightweight wake-word engines, KWS systems are consistently active, listening for a specific phrase to trigger further actions. As depicted in the figure, when we say "OK, Google" or "Alexa," this initiates a process on a microcontroller embedded within the device. Despite their limited resources, these microcontrollers play an important role in enabling seamless voice interactions with devices, often operating in environments with high ambient noise. The uniqueness of the wake word, as shown in the figure, helps minimize false positives, ensuring that the system is not triggered inadvertently.

Figure 5.3: Keyword Spotting example: interacting with Alexa. Source: Amazon.

It is important to appreciate that these keyword-spotting technologies are not isolated; they integrate seamlessly into larger systems, processing signals continuously while managing low power consumption. These systems extend beyond simple keyword recognition, evolving to facilitate diverse sound detections, such as glass breaking. This evolution is geared towards creating intelligent devices capable of understanding and responding to vocal commands, heralding a future where even household appliances can be controlled through voice interactions.

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. The long-tail distribution of languages implies that many languages have limited data, making the development of supportive technologies challenging.

This level of accuracy and robustness hinges on the availability and quality of data, the ability to label the data correctly, and the transparency of the data for the end user before it is used to train the model. However, it all begins with clearly understanding the problem statement or definition.

Generally, in ML, problem definition has a few key steps:

1. Identifying the problem definition clearly
2. Setting clear objectives
3. Establishing success benchmark
4. Understanding end-user engagement/use
5. Understanding the constraints and limitations of deployment
6. Followed by finally doing the data collection.

A solid project foundation is essential for its trajectory and eventual success. Central to this foundation is first identifying a clear problem, such as ensuring that voice commands in voice assistance systems are recognized consistently across varying environments. Clear objectives, like creating representative datasets for diverse scenarios, provide a unified direction. Benchmarks, such as system accuracy in keyword detection, offer measurable outcomes to gauge progress. Engaging with stakeholders, from end-users to investors, provides invaluable insights and ensures alignment with market needs. Additionally, understanding platform constraints is important when exploring areas like voice assistance. Embedded systems, such as microcontrollers, come with inherent processing power, memory, and energy efficiency limitations. Recognizing these limitations ensures that functionalities, like keyword detection, are tailored to operate optimally, balancing performance with resource conservation.

In this context, using KWS as an example, we can break each of the steps out as follows:

1. **Identifying the Problem:** At its core, 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.

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.
    - False Positive Rate: The percentage of non-keywords incorrectly identified as keywords.
    - 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 16KB to fit in the always-on island 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:

   - Variety of Accents: Collect data from speakers with various accents to ensure wide-ranging recognition.
   - Background Noises: Include data samples with different ambient noises to train the model for real-world scenarios.
   - Keyword Variations: People might either pronounce keywords differently or have slight variations in the wake word itself. Ensure the dataset captures these nuances.

7. **Iterative Feedback and Refinement:** Once a prototype KWS system is developed, it's crucial to test it in real-world scenarios, gather feedback, and iteratively refine the model. This ensures that the system remains aligned with the defined problem and objectives. This is important because the deployment scenarios change over time as things evolve.

Caution 4: Keyword Spotting with TensorFlow Lite Micro

Explore a hands-on guide for building and deploying Keyword Spotting systems using TensorFlow Lite Micro. Follow steps from data collection to model training and deployment to microcontrollers. Learn to create efficient KWS models that recognize specific keywords amidst background noise. Perfect for those interested in machine learning on embedded systems. Unlock the potential of voice-enabled devices with TensorFlow Lite Micro!

Open in Colab

The current chapter underscores the essential role of data quality in ML, using Keyword Spotting systems as an example. It outlines key steps, from problem definition to stakeholder engagement, emphasizing iterative feedback. The forthcoming chapter will dig deeper into data quality management, discussing its consequences and future trends, focusing on the importance of high-quality, diverse data in

AI system development, addressing ethical considerations and data sourcing methods.

5.3 Data Sourcing

The quality and diversity of data gathered are important for developing accurate and robust AI systems. Sourcing high-quality training data requires careful consideration of the objectives, resources, and ethical implications. Data can be obtained from various sources depending on the needs of the project:

5.3.1 Pre-existing datasets

Platforms like Kaggle and UCI Machine Learning Repository provide a convenient starting point. Pre-existing datasets are valuable for researchers, developers, and businesses. One of their primary advantages is cost efficiency. Creating a dataset from scratch can be time-consuming and expensive, so accessing ready-made data can save significant resources. Moreover, many datasets, like ImageNet, have become standard benchmarks in the machine learning community, allowing for consistent performance comparisons across different models and algorithms. This data availability means that experiments can be started immediately without any data collection and preprocessing delays. In a fast-moving field like ML, this practicality is important.

The quality assurance that comes with popular pre-existing datasets is important to consider because several datasets have errors in them. For instance, the ImageNet dataset was found to have over 6.4% errors. Given their widespread use, the community often identifies and rectifies any errors or biases in these datasets. This assurance is especially beneficial for students and newcomers to the field, as they can focus on learning and experimentation without worrying about data integrity. Supporting documentation often accompanying existing datasets is invaluable, though this generally applies only to 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. 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 platforms like Kaggle and UCI Machine Learning Repository 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, as multiple models might have been trained on them, leading to inflated performance metrics. Sometimes, these datasets do not reflect the real-world data.

In recent years, there has been growing awareness of bias, validity, and reproducibility issues that may exist in machine learning datasets. Figure 5.4 illustrates another critical concern: the potential for misalignment when using the same dataset to train different models.

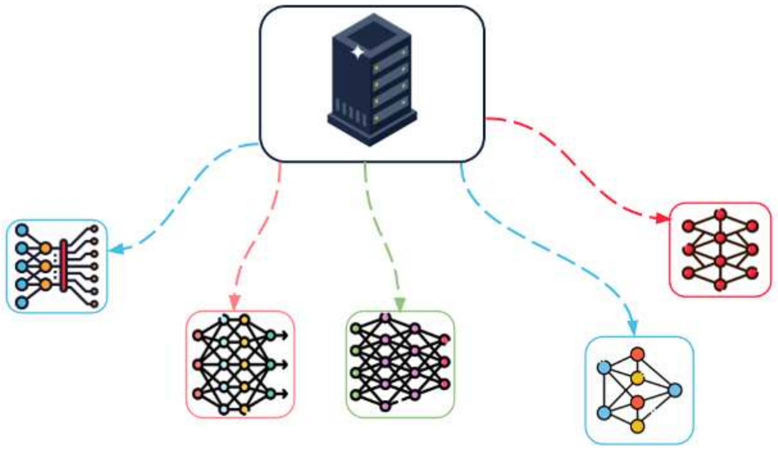


Figure 5.4: Training different models on the same dataset. Source: (icons from left to right: Becris; Freepik; Freepik; Paul J; SBTS2018).

As shown in Figure 5.4, training multiple models using the same dataset can result in a 'misalignment' between the models and the world. This misalignment creates an entire ecosystem of models that reflects only a narrow subset of the real-world data. Such a scenario can lead to limited generalization and potentially biased outcomes across various applications using these models.

5.3.2 Web Scraping

Web scraping refers to automated techniques for extracting data from websites. It typically involves sending HTTP requests to web servers, retrieving HTML content, and parsing that content to extract relevant information. Popular tools and frameworks for web scraping include Beautiful Soup, Scrapy, and Selenium. These tools offer different functionalities, from parsing HTML content to automating web browser interactions, especially for websites that load content dynamically using JavaScript.

Web scraping can effectively gather large datasets for training machine learning models, particularly when human-labeled data is scarce. For computer vision research, web scraping enables the collection of massive volumes of images and videos. Researchers have used this technique to build influential datasets like ImageNet and OpenImages. For example, one could scrape e-commerce sites to amass product photos for object recognition or social media platforms to collect user uploads for facial analysis. Even before ImageNet, Stanford's LabelMe project scraped Flickr for over 63,000 annotated images covering hundreds of object categories.

Beyond computer vision, web scraping supports gathering textual data for natural language tasks. Researchers can scrape news sites for sentiment analysis data, forums and review sites for dialogue systems research, or social media for topic modeling. For example, the training data for chatbot ChatGPT was obtained by scraping much of the public Internet. GitHub repositories were scraped to train GitHub's Copilot AI coding assistant.

Web scraping can also collect structured data, such as stock prices, weather data, or product information, for analytical applications. Once data is scraped, it is essential to store it in a structured manner, often using databases or data warehouses. Proper data management ensures the usability of the scraped data for future analysis and applications.

However, while web scraping offers numerous advantages, there are significant limitations and ethical considerations to bear. Not all websites permit scraping, and violating these restrictions can lead to legal repercussions. Scraping copyrighted material or private communications is also unethical and potentially illegal. Ethical web scraping mandates adherence to a website's 'robots.txt' file, which outlines the sections of the site that can be accessed and scraped by automated bots.

To deter automated scraping, many websites implement rate limits. If a bot sends too many requests in a short period, it might be temporarily blocked, restricting the speed of data access. Additionally, the dynamic nature of web content means that data scraped at different intervals might need more consistency, posing challenges for longitudinal studies. However, there are emerging trends like Web Navigation where machine learning algorithms can automatically navigate the website to access the dynamic content.

The volume of pertinent data available for scraping might be limited for niche subjects. For example, while scraping for common topics like images of cats and dogs might yield abundant data, searching for rare medical conditions might be less fruitful. Moreover, the data obtained through scraping is often unstructured and noisy, necessitating thorough preprocessing and cleaning. It is crucial to understand that not all scraped data will be of high quality or accuracy. Employing verification methods, such as cross-referencing with alternate data sources, can improve data reliability.

Privacy concerns arise when scraping personal data, emphasizing the need for anonymization. Therefore, it is paramount to adhere to a website's Terms of Service, confine data collection to public domains, and ensure the anonymity of any personal data acquired.

While web scraping can be a scalable method to amass large training datasets for AI systems, its applicability is confined to specific data types. For example, web scraping makes sourcing data for Inertial Measurement Units (IMU) for gesture recognition more complex. At most, one can scrape an existing dataset.

Web scraping can yield inconsistent or inaccurate data. For example, the photo in Figure 5.5 shows up when you search for 'traffic light' on Google Images. It is an image from 1914 that shows outdated traffic lights, which are also barely discernible because of the image's poor quality. This can be problematic for web-scraped datasets, as it pollutes the dataset with inapplicable (old) data samples.

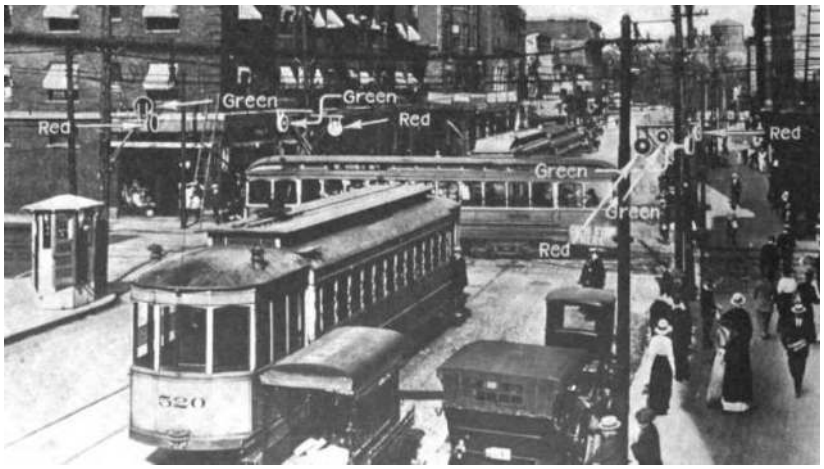


Figure 5.5: A picture of old traffic lights (1914). Source: Vox.

Caution 5: Web Scraping

Discover the power of web scraping with Python using libraries like Beautiful Soup and Pandas. This exercise will scrape Python documentation for function names and descriptions and explore NBA player stats. By the end, you'll have the skills to extract and analyze data from real-world websites. Ready to dive in? Access

the Google Colab notebook below and start practicing!
Open in Colab

5.3.3 Crowdsourcing

Crowdsourcing for datasets is the practice of obtaining data using the services of many people, either from a specific community or the general public, typically via the Internet. Instead of relying on a small team or specific organization to collect or label data, crowdsourcing leverages the collective effort of a vast, distributed group of participants. Services like Amazon Mechanical Turk enable the distribution of annotation tasks to a large, diverse workforce. This facilitates the collection of labels for complex tasks like sentiment analysis or image recognition requiring human judgment.

Crowdsourcing has emerged as an effective approach for data collection and problem-solving. One major advantage of crowdsourcing is scalability—by distributing tasks to a large, global pool of contributors on digital platforms, projects can process huge volumes of data quickly. This makes crowdsourcing ideal for large-scale data labeling, collection, and analysis.

In addition, crowdsourcing taps into a diverse group of participants, bringing a wide range of perspectives, cultural insights, and language abilities that can enrich data and enhance creative problem-solving in ways that a more homogenous group may not. Because crowdsourcing draws from a large audience beyond traditional channels, it is more cost-effective than conventional methods, especially for simpler microtasks.

Crowdsourcing platforms also allow for great flexibility, as task parameters can be adjusted in real time based on initial results. This creates a feedback loop for iterative improvements to the data collection process. Complex jobs can be broken down into microtasks and distributed to multiple people, with results cross-validated by assigning redundant versions of the same task. When thoughtfully managed, crowdsourcing enables community engagement around a collaborative project, where participants find reward in contributing.

However, while crowdsourcing offers numerous advantages, it's essential to approach it with a clear strategy. While it provides access to a diverse set of annotators, it also introduces variability in the quality of annotations. Additionally, platforms like Mechanical Turk might not always capture a complete demographic spectrum; often, tech-savvy individuals are overrepresented, while children and older people may be underrepresented. Providing clear instructions and training for the annotators is crucial. Periodic checks and validations of the labeled data help maintain quality. This ties back to the topic of clear Problem Definition that we discussed earlier. Crowdsourcing for datasets also requires careful attention to ethical considerations. It's crucial to ensure that participants are informed about how their data will be used and that their privacy is protected. Quality control through detailed protocols, transparency in sourcing, and auditing is essential to ensure reliable outcomes.

For TinyML, crowdsourcing can pose some unique challenges. TinyML devices are highly specialized for particular tasks within tight constraints. As a result, the data they require tends to be very specific. Obtaining such specialized data from a general audience may be difficult through crowdsourcing. For example, TinyML applications often rely on data collected from certain sensors or hardware. Crowdsourcing would require participants to have access to very specific and consistent devices - like microphones, with the same sampling rates. These hardware nuances present obstacles even for simple audio tasks like keyword spotting.

Beyond hardware, the data itself needs high granularity and quality, given the limitations of TinyML. It can be hard to ensure this when crowdsourcing from those unfamiliar with the application's context and requirements. There are also potential issues around privacy, real-time collection, standardization, and technical expertise. Moreover, the narrow nature of many TinyML tasks makes accurate data labeling easier with the proper understanding. Participants may need full context to provide reliable annotations.

Thus, while crowdsourcing can work well in many cases, the specialized needs of TinyML introduce unique data challenges. Careful planning is required for guidelines, targeting, and quality control. For some applications, crowdsourcing may be feasible, but others may require more focused data collection efforts to obtain relevant, high-quality training data.

5.3.4 Synthetic Data

Synthetic data generation can be a valuable solution for addressing data collection limitations. Figure 5.6 illustrates how this process works: synthetic data is merged with historical data to create a larger, more diverse dataset for model training.

As shown in the figure, synthetic data involves creating information that wasn't originally captured or observed but is generated using algorithms, simulations, or other techniques to resemble real-world

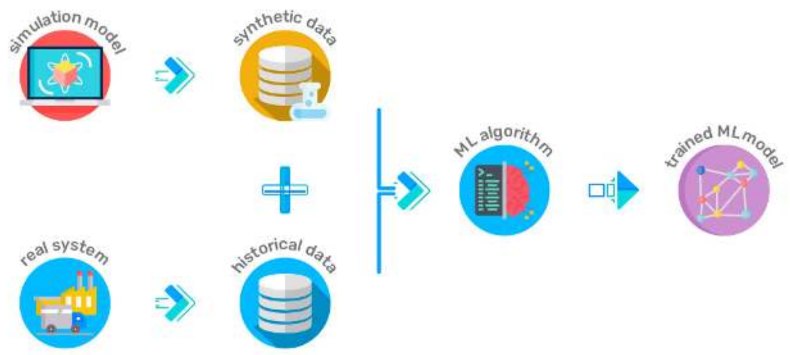


Figure 5.6: Increasing training data size with synthetic data generation. Source: AnyLogic.

data. This approach has become particularly valuable in fields where real-world data is scarce, expensive, or ethically challenging to obtain, such as in TinyML applications. Various techniques, including Generative Adversarial Networks (GANs), can produce high-quality synthetic data almost indistinguishable from real data. These methods have advanced significantly, making synthetic data generation increasingly realistic and reliable.

More real-world data may need to be available for analysis or training machine learning models in many domains, especially emerging ones. Synthetic data can fill this gap by producing large volumes of data that mimic real-world scenarios. For instance, detecting the sound of breaking glass might be challenging in security applications where a TinyML device is trying to identify break-ins. Collecting real-world data would require breaking numerous windows, which is impractical and costly.

Moreover, having a diverse dataset is crucial in machine learning, especially in deep learning. Synthetic data can augment existing datasets by introducing variations, thereby enhancing the robustness of models. For example, SpecAugment is an excellent data augmentation technique for Automatic Speech Recognition (ASR) systems.

Privacy and confidentiality are also big issues. Datasets containing sensitive or personal information pose privacy concerns when shared or used. Synthetic data, being artificially generated, doesn't have these direct ties to real individuals, allowing for safer use while preserving essential statistical properties.

Generating synthetic data, especially once the generation mechanisms have been established, can be a more cost-effective alternative. Synthetic data eliminates the need to break multiple windows to gather relevant data in the security above application scenario.

Many embedded use cases deal with unique situations, such as manufacturing plants, that are difficult to simulate. Synthetic data allows researchers complete control over the data generation process, enabling the creation of specific scenarios or conditions that are challenging to capture in real life.

While synthetic data offers numerous advantages, it is essential to use it judiciously. Care must be taken to ensure that the generated data accurately represents the underlying real-world distributions and does not introduce unintended biases.

Caution 6: Synthetic Data

Let us learn about synthetic data generation using Generative Adversarial Networks (GANs) on tabular data. We'll take a hands-on approach, diving into the workings of the CTGAN model and applying it to the Synthea dataset from the healthcare domain. From data preprocessing to model training and evaluation, we'll go step-by-step, learning how to create synthetic data, assess its quality, and unlock the potential of GANs for data augmentation and real-world applications.

Open in Colab

5.4 Data Storage

Data sourcing and data storage go hand in hand, and data must be stored in a format that facilitates easy access and processing. Depending on the use case, various kinds of data storage systems can be used to store your datasets. Some examples are shown in Table 5.1.

Table 5.1: Comparative overview of the database, data warehouse, and data lake.

Database	Data Warehouse	Data Lake
Purpose	Operational and transactional	Analytical
Data type	Structured	Structured, semi-structured, and/or unstructured

Database	Data Warehouse	Data Lake
Scale	Small to large volumes of data	Large volumes of integrated data Large volumes of diverse data
Examples	MySQL	Google BigQuery, Amazon Redshift, Microsoft Azure Synapse, Google Cloud Storage, AWS S3, Azure Data Lake Storage

The stored data is often accompanied by metadata, defined as 'data about data. It provides detailed contextual information about the data, such as means of data creation, time of creation, attached data use license, etc. Figure 5.7 illustrates the key pillars of data collection and their associated methods, highlighting the importance of structured data management. For example, Hugging Face has implemented Dataset Cards to promote responsible data use. These cards, which align with the documentation pillar shown in Figure 5.7, allow dataset creators to disclose potential biases and educate users about a dataset's contents and limitations.

The dataset cards provide important context on appropriate dataset usage by highlighting biases and other important details. Having this type of structured metadata can also allow for fast retrieval, aligning with the efficient data management principles illustrated in the figure. Once the model is developed and deployed to edge devices, the storage systems can continue to store incoming data, model updates, or analytical results, potentially utilizing methods from multiple pillars shown in Figure 5.7. This ongoing data collection and management process ensures that the model remains up-to-date and relevant in its operational environment.

Data Governance: With a large amount of data storage, it is also imperative to have policies and practices (i.e., data governance) that help manage data during its life cycle, from acquisition to disposal. Data governance outlines how data is managed and includes making key decisions about data access and control. Figure 5.8 illustrates the different domains involved in data governance. It involves exercising authority and making decisions concerning data to uphold its quality, ensure compliance, maintain security, and derive value. Data governance is operationalized by developing policies, incentives, and penalties, cultivating a culture that perceives data as a valuable asset. Specific procedures and assigned authorities are implemented

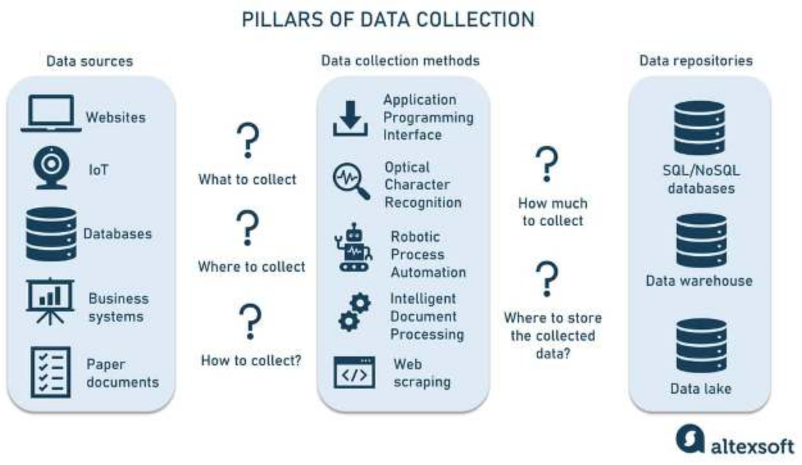


Figure 5.7: Pillars of data collection. Source: Alexsoft

to safeguard data quality and monitor its utilization and related risks.

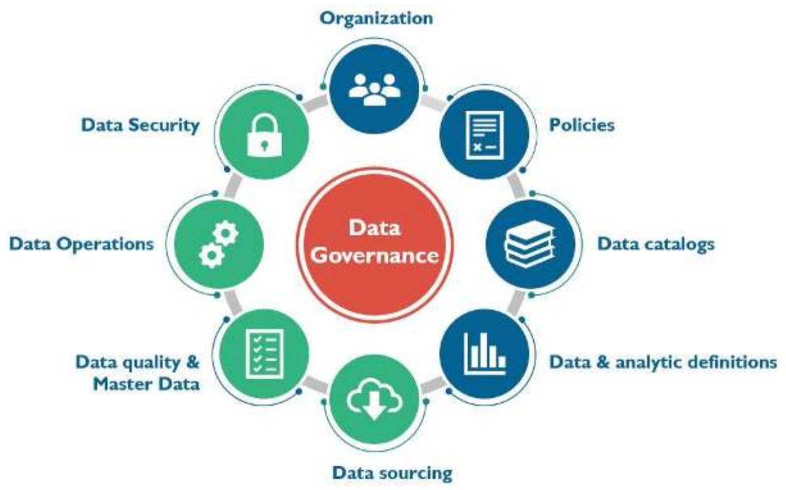


Figure 5.8: An overview of the data governance framework. Source: StarCIO..

Data governance utilizes three integrative approaches: planning and control, organizational, and risk-based.

- **The planning and control approach**, common in IT, aligns business and technology through annual cycles and continuous adjustments, focusing on policy-driven, auditable governance.
- **The organizational approach** emphasizes structure, establishing authoritative roles like Chief Data Officers and ensuring responsibility and accountability in governance.

- **The risk-based approach**, intensified by AI advancements, focuses on identifying and managing inherent risks in data and algorithms. It especially addresses AI-specific issues through regular assessments and proactive risk management strategies, allowing for incidental and preventive actions to mitigate undesired algorithm impacts.

Some examples of data governance across different sectors include:

- **Medicine:** Health Information Exchanges(HIEs) enable the sharing of health information across different healthcare providers to improve patient care. They implement strict data governance practices to maintain data accuracy, integrity, privacy, and security, complying with regulations such as the Health Insurance Portability and Accountability Act (HIPAA). Governance policies ensure that patient data is only shared with authorized entities and that patients can control access to their information.

- **Finance:** Basel III Framework is an international regulatory framework for banks. It ensures that banks establish clear policies, practices, and responsibilities for data management, ensuring data accuracy, completeness, and timeliness. Not only does it enable banks to meet regulatory compliance, but it also prevents financial crises by more effectively managing risks.

- **Government:** Government agencies managing citizen data, public records, and administrative information implement data governance to manage data transparently and securely. The Social Security System in the US and the Aadhar system in India are good examples of such governance systems.

Special data storage considerations for TinyML

Efficient Audio Storage Formats: Keyword spotting systems need specialized audio storage formats to enable quick keyword searching in audio data. Traditional formats like WAV and MP3 store full audio waveforms, which require extensive processing to search through. Keyword spotting uses compressed storage optimized for snippet-based search. One approach is to store compact acoustic features instead of raw audio. Such a workflow would involve:

- **Extracting acoustic features:** Mel-frequency cepstral coefficients (MFCCs) commonly represent important audio characteristics.

- **Creating Embeddings:** Embeddings transform extracted acoustic features into continuous vector spaces, enabling more compact and representative data storage. This representation is essential in converting high-dimensional data, like audio, into a more manageable and efficient format for computation and storage.

- **Vector quantization:** This technique represents high-dimensional data, like embeddings, with lower-dimensional vectors, reducing storage needs. Initially, a codebook is generated from the training data to define a set of code vectors representing the original data vectors. Subsequently, each data vector is matched to the nearest codeword according to the codebook, ensuring minimal information loss.

- **Sequential storage:** The audio is fragmented into short frames, and the quantized features (or embeddings) for each frame are stored sequentially to maintain the temporal order, preserving the coherence and context of the audio data.

This format enables decoding the features frame-by-frame for keyword matching. Searching the features is faster than decompressing the full audio.

Selective Network Output Storage: Another technique for reducing storage is to discard the intermediate audio features stored during training but not required during inference. The network is run on full audio during training. However, only the final outputs are stored during inference.

5.5 Data Processing

Data processing refers to the steps involved in transforming raw data into a format suitable for feeding into machine learning algorithms. It is a crucial stage in any ML workflow, yet often overlooked. With proper data processing, ML models are likely to achieve optimal performance. Figure 5.9 shows a breakdown of a data scientist's time allocation, highlighting the significant portion spent on data cleaning and organizing (%60).

Proper data cleaning is a crucial step that directly impacts model performance. Real-world data is often dirty, containing errors, missing values, noise, anomalies, and inconsistencies. Data cleaning involves detecting and fixing these issues to prepare high-quality data for modeling. By carefully selecting appropriate techniques, data scientists can improve model accuracy, reduce overfitting, and train algorithms to learn more robust patterns. Overall, thoughtful data processing allows machine learning systems to uncover insights better and make predictions from real-world data.

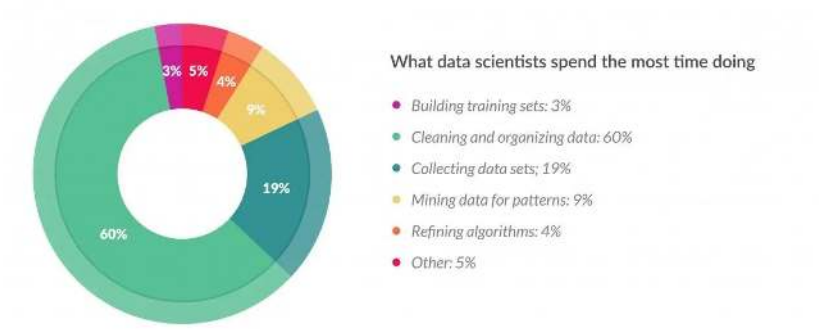


Figure 5.9: Data scientists' tasks breakdown by time spent. Source: Forbes.

Data often comes from diverse sources and can be unstructured or semi-structured. Thus, processing and standardizing it is essential, ensuring it adheres to a uniform format. Such transformations may include:

- Normalizing numerical variables
- Encoding categorical variables
- Using techniques like dimensionality reduction

Data validation serves a broader role than ensuring adherence to certain standards, like preventing temperature values from falling below absolute zero. These issues arise in TinyML because sensors may malfunction or temporarily produce incorrect readings; such transients are not uncommon. Therefore, it is imperative to catch data errors early before propagating through the data pipeline. Rigorous validation processes, including verifying the initial annotation practices, detecting outliers, and handling missing values through techniques like mean imputation, contribute directly to the quality of datasets. This, in turn, impacts the performance, fairness, and safety of the models trained on them.

Let's take a look at Figure 5.10 for an example of a data processing pipeline. In the context of TinyML, the Multilingual Spoken Words Corpus (MSWC) is an example of data processing pipelines—systematic and automated workflows for data transformation, storage, and processing. The input data (which's a collection of short recordings) goes through several phases of processing, such as audio-word alignement and keyword extraction.

MSWC streamlines the data flow, from raw data to usable datasets, data pipelines improve productivity and facilitate the rapid development of machine learning models. The MSWC is an expansive and expanding collection of audio recordings of spoken words in 50 different languages, which are collectively used by over 5 billion people. This dataset is intended for academic study and business uses in areas like keyword identification and speech-based search. It is openly licensed under Creative Commons Attribution 4.0 for broad usage.

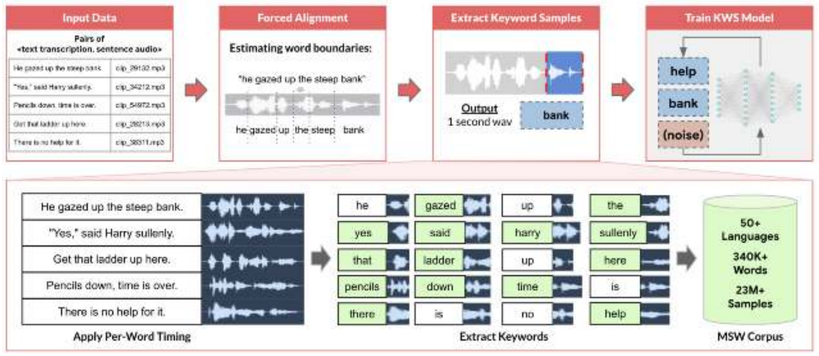

Input Data
Pairs of «text transcription, sentence audio»
He gazed up the steep bank. | clip_29132.mp3
"Yes," said Harry sullenly. | clip_34212.mp3
Pencils down, time is over. | clip_54972.mp3
Get that ladder up here. | clip_28213.mp3
There is no help for it. | clip_38311.mp3
Forced Alignment
Estimating 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 5.10: An overview of the Multilingual Spoken Words Corpus (MSWC) data processing pipeline. Source: Mazumder et al. (2021).

The MSWC used a forced alignment method to automatically extract individual word recordings to train keyword-spotting models from the Common Voice project, which features crowdsourced sentence-level recordings. Forced alignment refers to long-standing methods in speech processing that predict when speech phenomena like syllables, words, or sentences start and end within an audio recording. In the MSWC data, crowdsourced recordings often feature background noises, such as static and wind. Depending on the model's requirements, these noises can be removed or intentionally retained.

Maintaining the integrity of the data infrastructure is a continuous endeavor. This encompasses data storage, security, error handling, and stringent version control. Periodic updates are crucial, especially in dynamic realms like keyword spotting, to adjust to evolving linguistic trends and device integrations.

There is a boom in data processing pipelines, commonly found in ML operations toolchains, which we will discuss in the MLOps chapter. Briefly, these include frameworks like MLOps by Google Cloud. It provides methods for automation and monitoring at all steps of ML system construction, including integration, testing, releasing, deployment, and infrastructure management. Several mechanisms focus on data processing, an integral part of these systems.

Caution 7: Data Processing

Let us explore two significant projects in speech data processing and machine learning. The MSWC is a vast audio dataset with over 340,000 keywords and 23.4 million 1-second spoken examples. It's used in various applications like voice-enabled devices and call center automation. The Few-Shot Keyword Spotting project introduces a new approach for keyword spotting across different languages, achieving impressive results with minimal training data. We'll look into the MSWC dataset, learn how to structure it effectively, and then train a few-shot keyword-spotting model. Let's get started!

Open in Colab

5.6 Data Labeling

Data labeling is important in creating high-quality training datasets for machine learning models. Labels provide ground truth information, allowing models to learn relationships between inputs and desired outputs. This section covers key considerations for selecting label types, formats, and content to capture the necessary information for tasks. It discusses common annotation approaches, from manual labeling to crowdsourcing to AI-assisted methods, and best practices for ensuring label quality through training, guidelines, and quality checks. We also emphasize the ethical treatment of human annotators. The integration of AI to accelerate and augment human annotation is also explored. Understanding labeling needs, challenges, and strategies are essential for constructing reliable, useful datasets to train performant, trustworthy machine learning systems.

5.6.1 Label Types

Labels capture information about key tasks or concepts. Figure 5.11 includes some common label types: a "classification label" is used for categorizing images with labels (labeling an image with "dog" if it features a dog); a "bounding box" identifies object location (drawing a box around the dog); a "segmentation map" classifies objects at the pixel level (highlighting the dog in a distinct color); a "caption" provides descriptive annotations (describing the dog's actions, position, color, etc.); and a "transcript" denotes audio content. The choice of label format depends on the use case and resource constraints, as more detailed labels require greater effort to collect (Johnson-Roberson et al. 2017).

Unless focused on self-supervised learning, a dataset will likely provide labels addressing one or more tasks of interest. Given their unique resource constraints, dataset creators must consider what

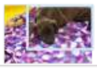
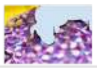
Label Type	Input Type	Output Type
Classification Label		"Dog", "Blanket", "No cat"
Bounding Box		
Segmentation Map		
Caption		"A dog curls up on a spotted purple blanket."
Transcript	🔊	"Once upon a time, a dog was curled up on a spotted purple blanket …"

Figure 5.11: An overview of common label types.

information labels should capture and how they can practically obtain the necessary labels. Creators must first decide what type(s) of content labels should capture. For example, a creator interested in car detection would want to label cars in their dataset. Still, they might also consider whether to simultaneously collect labels for other tasks that the dataset could potentially be used for, such as pedestrian detection.

Additionally, annotators can provide metadata that provides insight into how the dataset represents different characteristics of interest (see Section 5.9). The Common Voice dataset, for example, includes various types of metadata that provide information about the speakers, recordings, and dataset quality for each language represented (Ardila et al. 2020). They include demographic splits showing the number of recordings by speaker age range and gender. This allows us to see who contributed recordings for each language. They also include statistics like average recording duration and total hours of validated recordings. These give insights into the nature and size of the datasets for each language.

Additionally, quality control metrics like the percentage of recordings that have been validated are useful to know how complete and clean the datasets are. The metadata also includes normalized demographic splits scaled to 100% for comparison across languages. This highlights representation differences between higher and lower resource languages.

Next, creators must determine the format of those labels. For example, a creator interested in car detection might choose between binary classification labels that say whether a car is present, bounding boxes that show the general locations of any cars, or pixel-wise segmentation labels that show the exact location of each car. Their choice of label format may depend on their use case and resource constraints, as finer-grained labels are typically more expensive and time-consuming to acquire.

5.6.2 Annotation Methods

Common annotation approaches include manual labeling, crowdsourcing, and semi-automated techniques. Manual labeling by experts yields high quality but needs more scalability. Crowdsourcing enables non-experts to distribute annotation, often through dedicated platforms (Sheng and Zhang 2019). Weakly supervised and programmatic methods can reduce manual effort by heuristically or automatically generating labels (Ratner et al. 2018).

After deciding on their labels' desired content and format, creators begin the annotation process. To collect large numbers of labels from human annotators, creators frequently rely on dedicated annotation platforms, which can connect them to teams of human annotators. When using these platforms, creators may need more insight into annotators' backgrounds and experience levels with topics of interest. However, some platforms offer access to annotators with specific expertise (e.g., doctors).

Caution 8: Bootstrapped Labels

Let us explore Wake Vision, a comprehensive dataset designed for TinyML person detection. This dataset is derived from a larger, general-purpose dataset, Open Images (Kuznetsova et al. 2020), and tailored specifically for binary person detection.
The transformation process involves filtering and relabeling the existing labels and bounding boxes in Open Images using an automated pipeline. This method not only conserves time and resources but also ensures the dataset meets the specific requirements of TinyML applications.
Additionally, we generate metadata to benchmark the fairness and robustness of models in challenging scenarios.
Let's get started!

Open in Colab

5.6.3 Ensuring Label Quality

There is no guarantee that the data labels are actually correct. Figure 5.12 shows some examples of hard labeling cases: some errors arise from blurred pictures that make them hard to identify (the frog image), and others stem from a lack of domain knowledge (the black stork case). It is possible that despite the best instructions being given to labelers, they still mislabel some images (Northcutt, Athalye, and Mueller 2021). Strategies like quality checks, training annotators, and collecting multiple labels per datapoint can help ensure label quality. For ambiguous tasks, multiple annotators can help identify controversial datapoints and quantify disagreement levels.

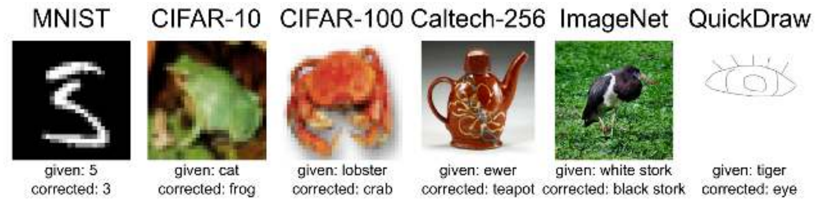


Figure 5.12: Some examples of hard labeling cases. Source: Northcutt, Athalye, and Mueller (2021).

When working with human annotators, offering fair compensation and otherwise prioritizing ethical treatment is important, as annotators can be exploited or otherwise harmed during the labeling process (Perrigo, 2023). For example, if a dataset is likely to contain disturbing content, annotators may benefit from having the option to view images in grayscale (Google, n.d.).

5.6.4 AI-Assisted Annotation

ML has an insatiable demand for data. Therefore, more data is needed. This raises the question of how we can get more labeled data. Rather than always generating and curating data manually, we can rely on existing AI models to help label datasets more quickly and cheaply, though often with lower quality than human annotation. This can be done in various ways as shown in Figure 5.13, including the following:

- **Pre-annotation:** AI models can generate preliminary labels for a dataset using methods such as semi-supervised learning (Chapelle, Scholkopf, and Zien 2009), which humans can then review and correct. This can save a significant amount of time, especially for large datasets.
- **Active learning:** AI models can identify the most informative data points in a dataset, which can then be prioritized for human annotation. This can help improve the labeled dataset's quality while reducing the overall annotation time.

- **Quality control:** AI models can identify and flag potential errors in human annotations, helping to ensure the accuracy and consistency of the labeled dataset.

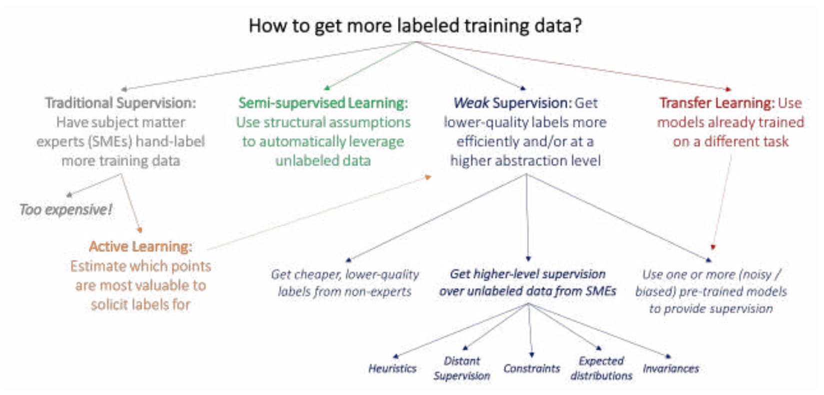

How to get more labeled training data?
Traditional Supervision: Have subject matter experts (SMEs) hand-label more training data
Semi-supervised Learning: Use structural assumptions to automatically leverage unlabeled data
Weak Supervision: Get lower-quality labels more efficiently and/or at a higher abstraction level
Transfer Learning: Use models already trained on a different task
Too expensive!
Active Learning: Estimate which points are most valuable to solicit labels for
Get cheaper, lower-quality labels from non-experts
Get higher-level supervision over unlabeled data from SMEs
Use one or more (noisy / biased) pre-trained models to provide supervision
Heuristics
Distant Supervision
Constraints
Expected distributions
Invariances

Figure 5.13: Strategies for acquiring additional labeled training data. Source: Stanford AI Lab.

Here are some examples of how AI-assisted annotation has been proposed to be useful:

- **Medical imaging:** AI-assisted annotation labels medical images, such as MRI scans and X-rays (R. Krishnan, Rajpurkar, and Topol 2022). Carefully annotating medical datasets is extremely challenging, especially at scale, since domain experts are scarce and become costly. This can help to train AI models to diagnose diseases and other medical conditions more accurately and efficiently.

- **Self-driving cars:** AI-assisted annotation is being used to label images and videos from self-driving cars. This can help to train AI models to identify objects on the road, such as other vehicles, pedestrians, and traffic signs.
- **Social media:** AI-assisted annotation labels social media posts like images and videos. This can help to train AI models to identify and classify different types of content, such as news, advertising, and personal posts.

5.7 Data Version Control

Production systems are perpetually inundated with fluctuating and escalating volumes of data, prompting the rapid emergence of numerous data replicas. This increasing data serves as the foundation for training machine learning models. For instance, a global sales company engaged in sales forecasting continuously receives consumer behavior data. Similarly, healthcare systems formulating predictive models for disease diagnosis are consistently acquiring new patient data. TinyML applications, such as keyword spotting, are highly data-hungry regarding the amount of data generated. Consequently, meticulous tracking of data versions and the corresponding model performance is imperative.

Data Version Control offers a structured methodology to handle alterations and versions of datasets efficiently. It facilitates monitoring modifications, preserves multiple versions, and guarantees reproducibility and traceability in data-centric projects. Furthermore, data version control provides the versatility to review and use specific versions as needed, ensuring that each stage of the data processing and model development can be revisited and audited precisely and easily. It has a variety of practical uses -

Risk Management: Data version control allows transparency and accountability by tracking dataset versions.

Collaboration and Efficiency: Easy access to different dataset versions in one place can improve data sharing of specific checkpoints and enable efficient collaboration.

Reproducibility: Data version control allows for tracking the performance of models concerning different versions of the data, and therefore enabling reproducibility.

Key Concepts

- **Commits:** It is an immutable snapshot of the data at a specific point in time, representing a unique version. Every commit is associated with a unique identifier to allow
- **Branches:** Branching allows developers and data scientists to diverge from the main development line and continue to work independently without affecting other branches. This is especially useful when experimenting with new features or models, enabling parallel development and experimentation without the risk of corrupting the stable main branch.
- **Merges:** Merges help to integrate changes from different branches while maintaining the integrity of the data.

With data version control in place, we can track the changes shown in Figure 5.14, reproduce previous results by reverting to older versions, and collaborate safely by branching off and isolating the changes.

Popular Data Version Control Systems

[DVC]: It stands for Data Version Control in short and is an open-source, lightweight tool that works on top of Git Hub and supports all

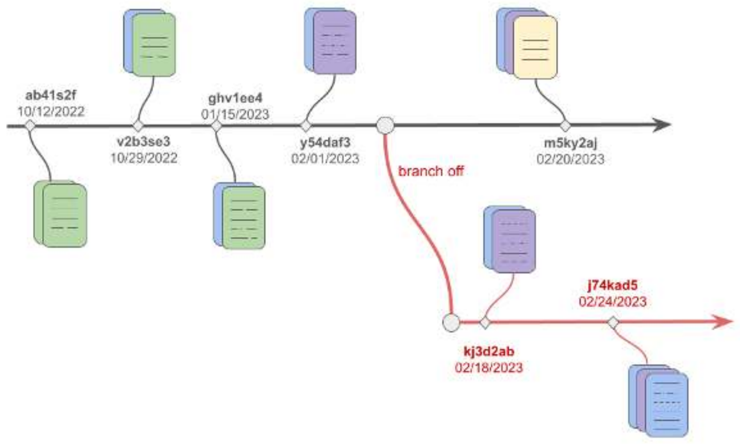


Figure 5.14: Data versioning.

kinds of data formats. It can seamlessly integrate into the workflow if Git is used to manage code. It captures the versions of data and models in the Git commits while storing them on-premises or on the cloud (e.g., AWS, Google Cloud, Azure). These data and models (e.g., ML artifacts) are defined in the metadata files, which get updated in every commit. It can allow metrics tracking of models on different versions of the data.

lakeFS: It is an open-source tool that supports the data version control on data lakes. It supports many git-like operations, such as branching and merging of data, as well as reverting to previous versions of the data. It also has a unique UI feature, making exploring and managing data much easier.

Git LFS: It is useful for data version control on smaller-sized datasets. It uses Git's inbuilt branching and merging features but is limited in tracking metrics, reverting to previous versions, or integrating with data lakes.

5.8 Optimizing Data for Embedded AI

Creators working on embedded systems may have unusual priorities when cleaning their datasets. On the one hand, models may be developed for unusually specific use cases, requiring heavy filtering of datasets. While other natural language models may be capable of turning any speech into text, a model for an embedded system may be focused on a single limited task, such as detecting a keyword. As a result, creators may aggressively filter out large amounts of data because they need to address the task of interest. An embedded AI system may also be tied to specific hardware devices or environments. For example, a video model may need to process images from a single type of camera, which will only be mounted on doorbells in residential neighborhoods. In this scenario, creators may discard images if they came from a different kind of camera, show the wrong type of scenery, or were taken from the wrong height or angle.

On the other hand, embedded AI systems are often expected to provide especially accurate performance in unpredictable real-world settings. This may lead creators to design datasets to represent variations in potential inputs and promote model robustness. As a result, they may define a narrow scope for their project but then aim for deep coverage within those bounds. For example, creators of the doorbell model mentioned above might try to cover variations in data arising from:

- Geographically, socially, and architecturally diverse neighborhoods
- Different types of artificial and natural lighting
- Different seasons and weather conditions
- Obstructions (e.g., raindrops or delivery boxes obscuring the camera's view)

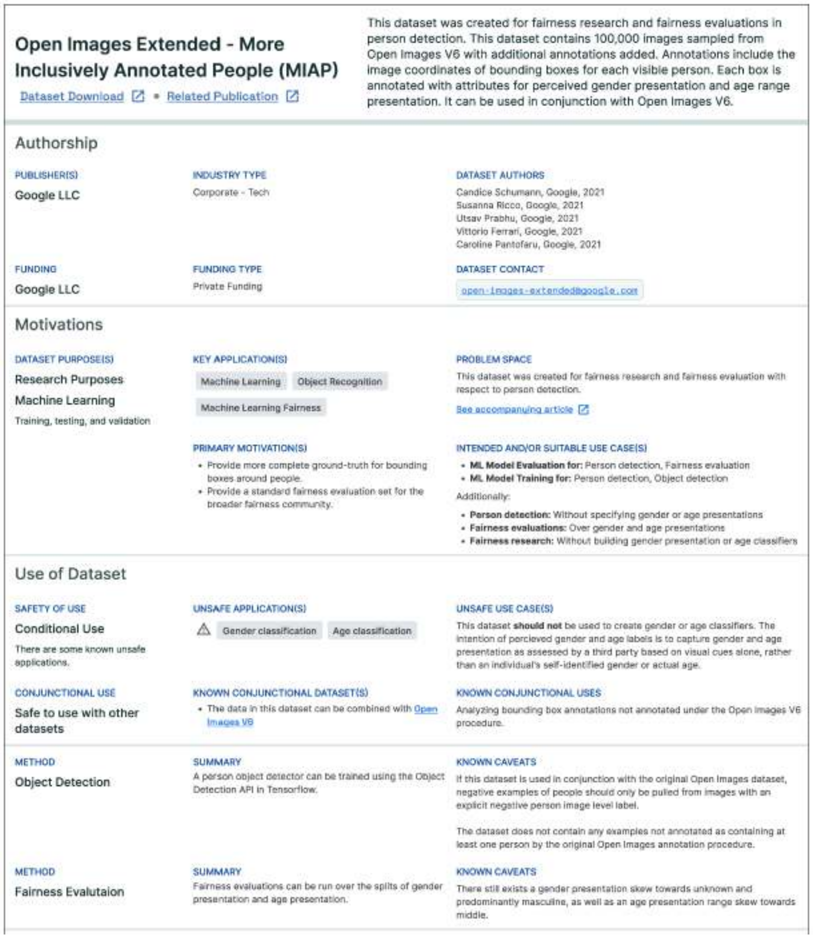
As described above, creators may consider crowdsourcing or synthetically generating data to include these variations.

5.9 Data Transparency

By providing clear, detailed documentation, creators can help developers understand how best to use their datasets. Several groups have suggested standardized documentation formats for datasets, such as Data Cards (Pushkarna, Zaldivar, and Kjartansson 2022), datasheets (Gebru et al. 2021), data statements (Bender and Friedman 2018), or Data Nutrition Labels (Holland et al. 2020). When releasing a dataset, creators may describe what kinds of data they collected, how they collected and labeled it, and what kinds of use cases may be a good or poor fit for the dataset. Quantitatively, it may be appropriate to show how well the dataset represents different groups (e.g., different gender groups, different cameras).

Figure 5.15 shows an example of a data card for a computer vision (CV) dataset. It includes some basic information about the dataset and instructions on how to use it, including known biases.

Keeping track of data provenance- essentially the origins and the journey of each data point through the data pipeline- is not merely a good practice but an essential requirement for data quality. Data provenance contributes significantly to the transparency of machine



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

PROBLEM SPACE
This dataset was created for fairness research and fairness evaluation with respect to person detection.
See accompanying article

PRIMARY MOTIVATION(S)
- Provide more complete ground-truth for bounding boxes around people.
- Provide a standard fairness evaluation set for the broader fairness community.

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 5.15: Data card describing a CV dataset. Source: Pushkarna, Zaldivar, and Kjartansson (2022).

learning systems. Transparent systems make it easier to scrutinize data points, enabling better identification and rectification of errors, biases, or inconsistencies. For instance, if an ML model trained on medical data is underperforming in particular areas, tracing the provenance can help identify whether the issue is with the data collection methods, the demographic groups represented in the data or other factors. This level of transparency doesn't just help debug the system but also plays a crucial role in enhancing the overall data quality. By improving the reliability and credibility of the dataset, data provenance also enhances the model's performance and its acceptability among end-users.

When producing documentation, creators should also specify how users can access the dataset and how the dataset will be maintained over time. For example, users may need to undergo training or receive special permission from the creators before accessing a protected information dataset, as with many medical datasets. In some cases, users may not access the data directly. Instead, they must submit their model to be trained on the dataset creators' hardware, following a federated learning setup (Aledhari et al. 2020). Creators may also describe how long the dataset will remain accessible, how the users can submit feedback on any errors they discover, and whether there are plans to update the dataset.

Some laws and regulations also promote data transparency through new requirements for organizations:

- General Data Protection Regulation (GDPR) in the European Union: It establishes strict requirements for processing and protecting the personal data of EU citizens. It mandates plain-language privacy policies that clearly explain what data is collected, why it is used, how long it is stored, and with whom it is shared. GDPR also mandates that privacy notices must include details on the legal basis for processing, data transfers, retention periods, rights to access and deletion, and contact info for data controllers.
- California's Consumer Privacy Act (CCPA): CCPA requires clear privacy policies and opt-out rights to sell personal data. Significantly, it also establishes rights for consumers to request their specific data be disclosed. Businesses must provide copies of collected personal information and details on what it is used for, what categories are collected, and what third parties receive. Consumers can identify data points they believe need to be more accurate. The law represents a major step forward in empowering personal data access.

Ensured data transparency presents several challenges, especially because it requires significant time and financial resources. Data systems are also quite complex, and full transparency can take time. Full transparency may also overwhelm consumers with too much detail. Finally, it is also important to balance the tradeoff between transparency and privacy.

5.10 Licensing

Many high-quality datasets either come from proprietary sources or contain copyrighted information. This introduces licensing as a challenging legal domain. Companies eager to train ML systems must engage in negotiations to obtain licenses that grant legal access to these datasets. Furthermore, licensing terms can impose restrictions on data applications and sharing methods. Failure to comply with these licenses can have severe consequences.

For instance, ImageNet, one of the most extensively utilized datasets for computer vision research, is a case in point. Most of its images were procured from public online sources without explicit permission, sparking ethical concerns (Prabhu and Birhane, 2020). Accessing the ImageNet dataset for corporations requires registration and adherence to its terms of use, which restricts commercial usage (ImageNet, 2021). Major players like Google and Microsoft invest significantly in licensing datasets to improve their ML vision systems. However, the cost factor restricts accessibility for researchers from smaller companies with constrained budgets.

The legal domain of data licensing has seen major cases that help define fair use parameters. A prominent example is *Authors Guild, Inc. v. Google, Inc.* This 2005 lawsuit alleged that Google's book scanning project infringed copyrights by displaying snippets without permission. However, the courts ultimately ruled in Google's favor, upholding fair use based on the transformative nature of creating a searchable index and showing limited text excerpts. This precedent provides some legal grounds for arguing fair use protections apply to indexing datasets and generating representative samples for machine learning. However, license restrictions remain binding, so a comprehensive analysis of licensing terms is critical. The case demonstrates why negotiations with data providers are important to enable legal usage within acceptable bounds.

New Data Regulations and Their Implications

New data regulations also impact licensing practices. The legislative landscape is evolving with regulations like the EU's Artificial Intelligence Act, which is poised to regulate AI system development and use within the European Union (EU). This legislation:

1. Classifies AI systems by risk.
2. Mandates development and usage prerequisites.
3. Emphasizes data quality, transparency, human oversight, and accountability.

Additionally, the EU Act addresses the ethical dimensions and operational challenges in sectors such as healthcare and finance. Key elements include the prohibition of AI systems posing "unacceptable" risks, stringent conditions for high-risk systems, and minimal obligations for "limited risk" AI systems. The proposed European AI Board will oversee and ensure the implementation of efficient regulation.

Challenges in Assembling ML Training Datasets

Complex licensing issues around proprietary data, copyright law, and privacy regulations constrain options for assembling ML training datasets. However, expanding accessibility through more open licensing or public-private data collaborations could greatly accelerate industry progress and ethical standards.

Sometimes, certain portions of a dataset may need to be removed or obscured to comply with data usage agreements or protect sensitive information. For example, a dataset of user information may have names, contact details, and other identifying data that may need to be removed from the dataset; this is well after the dataset has already been actively sourced and used for training models. Similarly, a dataset that includes copyrighted content or trade secrets may need to filter out those portions before being distributed. Laws such as the General Data Protection Regulation (GDPR), the California Consumer Privacy Act (CCPA), and the Amended Act on the Protection of Personal Information (APPI) have been passed to guarantee the right to be forgotten. These regulations legally require model providers to erase user data upon request.

Data collectors and providers need to be able to take appropriate measures to de-identify or filter out any proprietary, licensed, confidential, or regulated information as needed. Sometimes, the users may explicitly request that their data be removed.

The ability to update the dataset by removing data from the dataset will enable the creators to uphold legal and ethical obligations around data usage and privacy. However, the ability to remove data has some important limitations. We must consider that some models may have already been trained on the dataset, and there is no clear or known way to eliminate a particular data sample's effect from the trained network. There is no erase mechanism. Thus, this begs the question, should the model be retrained from scratch each time a sample is removed? That's a costly option. Once data has been used to train a model, simply removing it from the original dataset may not fully eliminate its impact on the model's behavior. New research is needed around the effects of data removal on already-trained models and whether full retraining is necessary to avoid retaining artifacts of deleted data. This presents an important consideration when balancing data licensing obligations with efficiency and practicality in an evolving, deployed ML system.

Dataset licensing is a multifaceted domain that intersects technology, ethics, and law. Understanding these intricacies becomes paramount for anyone building datasets during data engineering as the world evolves.

5.11 Conclusion

Data is the fundamental building block of AI systems. Without quality data, even the most advanced machine learning algorithms will fail. Data engineering encompasses the end-to-end process of collecting, storing, processing, and managing data to fuel the development of machine learning models. It begins with clearly defining the core problem and objectives, which guides effective data collection. Data can be sourced from diverse means, including existing datasets, web scraping, crowdsourcing, and synthetic data generation. Each approach involves tradeoffs between cost, speed, privacy, and specificity. Once data is collected, thoughtful labeling through manual or AI-assisted annotation enables the creation of high-quality training datasets. Proper storage in databases, warehouses, or lakes facilitates easy access and analysis. Metadata provides contextual details about the data. Data processing transforms raw data into a clean, consistent format for machine learning model development. Throughout this pipeline, transparency through documentation and provenance tracking is crucial for ethics, auditability, and reproducibility. Data licensing protocols also govern legal data access and use. Key challenges in data engineering include privacy risks, representation gaps, legal restrictions around proprietary data, and the need to balance competing constraints like speed versus quality. By thoughtfully engineering high-quality training data, machine learning practitioners can develop accurate, robust, and responsible AI systems, including embedded and TinyML applications.

5.12 Resources

Here is a curated list of resources to support students and instructors in their learning and teaching journeys. We are continuously working

on expanding this collection and will add new exercises soon.

Slides

These slides are a valuable tool for instructors to deliver lectures and for students to review the material at their own pace. We encourage students and instructors to leverage these slides to improve their understanding and facilitate effective knowledge transfer.

- 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

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.

- Exercise 4
- Exercise 5
- Exercise 6

- Exercise 7
- Exercise 8

Chapter 6

AI Frameworks

Figure 6.1: *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.*

This chapter explores the landscape of AI frameworks that serve as the foundation for developing machine learning systems. AI frameworks provide the tools, libraries, and environments to design, train, and deploy machine learning models. We explore the evolutionary trajectory of these frameworks, dissect the workings of TensorFlow, and provide insights into the core components and advanced features that define these frameworks.

Furthermore, we investigate the specialization of frameworks tailored to specific needs, the emergence of frameworks specifically designed for embedded AI, and the criteria for selecting the most suitable framework for your project. This exploration will be rounded off by a glimpse into the future trends expected to shape the landscape of ML frameworks in the coming years.

Learning Objectives

- Understand the evolution and capabilities of major machine learning frameworks. This includes graph execution models, programming paradigms, hardware acceleration support, and how they have expanded over time.
- Learn frameworks' core components and functionality, such as computational graphs, data pipelines, optimization algorithms, training loops, etc., that enable efficient model building.
- Compare frameworks across different environments, such as cloud, edge, and TinyML. Learn how frameworks specialize based on computational constraints and hardware.
- Dive deeper into embedded and TinyML-focused frameworks like TensorFlow Lite Micro, CMSIS-NN, TinyEngine, etc., and how they optimize for microcontrollers.
- When choosing a framework, explore model conversion and deployment considerations, including latency, memory usage, and hardware support.
- Evaluate key factors in selecting the right framework, like performance, hardware compatibility, community support, ease of use, etc., based on the specific project needs and constraints.
- Understand the limitations of current frameworks and potential future trends, such as using ML to improve frameworks, decomposed ML systems, and high-performance compilers.

6.1 Overview

Machine learning frameworks provide the tools and infrastructure to efficiently build, train, and deploy machine learning models. In this chapter, we will explore the evolution and key capabilities of major frameworks like TensorFlow (TF), PyTorch, and specialized frameworks for embedded devices. We will dive into the components like computational graphs, optimization algorithms, hardware acceleration, and more that enable developers to construct performant models quickly. Understanding these frameworks is essential to leverage the power of deep learning across the spectrum from cloud to edge devices.

ML frameworks handle much of the complexity of model development through high-level APIs and domain-specific languages that allow practitioners to quickly construct models by combining pre-made components and abstractions. For example, frameworks like TensorFlow and PyTorch provide Python APIs to define neural network architectures using layers, optimizers, datasets, and more. This enables rapid iteration compared to coding every model detail from scratch.

A key capability offered by these frameworks is distributed training engines that can scale model training across clusters of GPUs and TPUs. This makes it feasible to train state-of-the-art models with billions or trillions of parameters on vast datasets. Frameworks also integrate with specialized hardware like NVIDIA GPUs to further accelerate training via optimizations like parallelization and efficient matrix operations.

In addition, frameworks simplify deploying finished models into production through tools like TensorFlow Serving for scalable model serving and TensorFlow Lite for optimization on mobile and edge devices. Other valuable capabilities include visualization, model optimization techniques like quantization and pruning, and monitoring metrics during training.

Leading open-source frameworks like TensorFlow, PyTorch, and MXNet power much of AI research and development today. Commercial offerings like Amazon SageMaker and Microsoft Azure Machine Learning integrate these open source frameworks with proprietary capabilities and enterprise tools.

Machine learning engineers and practitioners leverage these robust frameworks to focus on high-value tasks like model architecture, feature engineering, and hyperparameter tuning instead of infrastructure. The goal is to build and deploy performant models that solve real-world problems efficiently.

In this chapter, we will explore today's leading cloud frameworks and how they have adapted models and tools specifically for embedded and edge deployment. We will compare programming models, supported hardware, optimization capabilities, and more to fully understand how frameworks enable scalable machine learning from the cloud to the edge.

6.2 Framework Evolution

Machine learning frameworks have evolved significantly to meet the diverse needs of machine learning practitioners and advancements in AI techniques. A few decades ago, building and training machine learning models required extensive low-level coding and infrastructure. Alongside the need for low-level coding, early neural network research was constrained by insufficient data and computing power. However, machine learning frameworks have evolved considerably over the past decade to meet the expanding needs of practitioners and rapid advances in deep learning techniques. The release of large datasets like ImageNet (Deng et al. 2009) and advancements in parallel GPU computing unlocked the potential for far deeper neural networks.

The first ML frameworks, Theano by Team et al. (2016) and Caffe by Y. Jia et al. (2014), were developed by academic institutions. Theano was created by the Montreal Institute for Learning Algorithms, while Caffe was developed by the Berkeley Vision and Learning Center. Amid growing interest in deep learning due to state-of-the-art performance of AlexNet Krizhevsky, Sutskever, and Hinton (2012) on the ImageNet dataset, private companies and individuals began developing ML frameworks, resulting in frameworks such as Keras by Chollet (2018), Chainer by Tokui et al. (2019), TensorFlow from Google (Yu et al. 2018), CNTK by Microsoft (Seide and Agarwal 2016), and PyTorch by Facebook (Ansel et al. 2024).

Many of these ML frameworks can be divided into high-level vs. low-level frameworks and static vs. dynamic computational graph frameworks. High-level frameworks provide a higher level of abstraction than low-level frameworks. High-level frameworks have pre-built functions and modules for common ML tasks, such as creating, training, and evaluating common ML models, preprocessing data, engineering features, and visualizing data, which low-level frameworks do not have. Thus, high-level frameworks may be easier to use but are less customizable than low-level frameworks (i.e., users of low-level frameworks can define custom layers, loss functions, optimization algorithms, etc.). Examples of high-level frameworks include TensorFlow/Keras and PyTorch. Examples of low-level ML frameworks include TensorFlow with low-level APIs, Theano, Caffe, Chainer, and CNTK.

Frameworks like Theano and Caffe used static computational graphs, which required defining the full model architecture upfront, thus limiting flexibility. In contract, dynamic graphs are constructed on the fly for more iterative development. Around 2016, frameworks like PyTorch and TensorFlow 2.0 began adopting dynamic graphs, providing greater flexibility for model development. We will discuss these concepts and details later in the AI Training section.

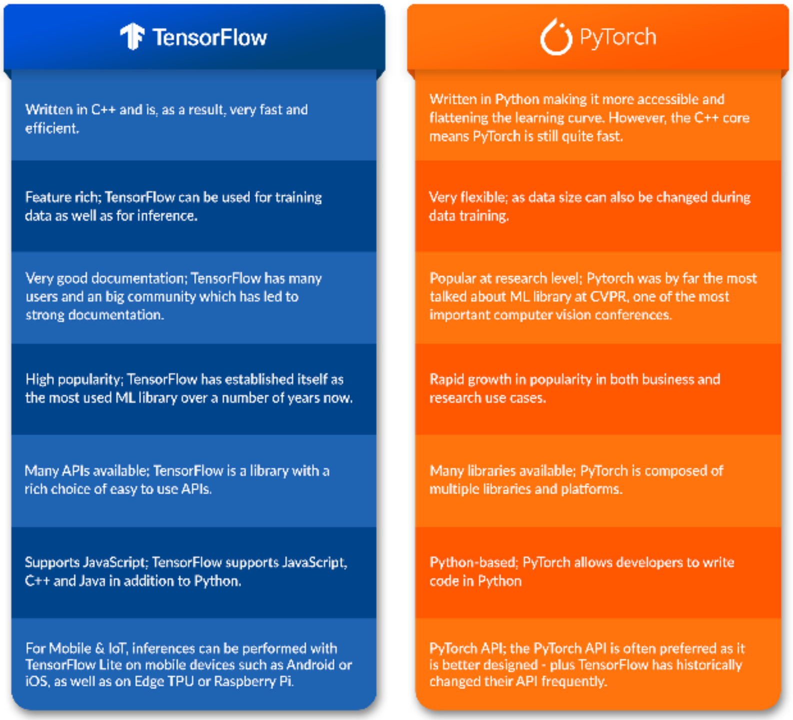
The development of these frameworks facilitated an explosion in model size and complexity over time—from early multilayer perceptrons and convolutional networks to modern transformers with billions or trillions of parameters. In 2016, ResNet models by K. He et al. (2016) achieved record ImageNet accuracy with over 150 layers and 25 million parameters. Then, in 2020, the GPT-3 language model from OpenAI (Brown et al. 2020) pushed parameters to an astonishing 175 billion using model parallelism in frameworks to train across thousands of GPUs and TPUs.

Each generation of frameworks unlocked new capabilities that powered advancement:

- Theano and TensorFlow (2015) introduced computational graphs and automatic differentiation to simplify model building.
- CNTK (2016) pioneered efficient distributed training by combining model and data parallelism.
- PyTorch (2016) provided imperative programming and dynamic graphs for flexible experimentation.
- TensorFlow 2.0 (2019) defaulted eager execution for intuitiveness and debugging.
- TensorFlow Graphics (2020) added 3D data structures to handle point clouds and meshes.

In recent years, the landscape of machine learning frameworks has significantly consolidated. Figure 6.3 illustrates this convergence, showing that TensorFlow and PyTorch have become the overwhelmingly dominant ML frameworks, collectively representing more than 95% of ML frameworks used in research and production. While both frameworks have risen to prominence, they have distinct characteristics. Figure 6.2 draws a contrast between the attributes of TensorFlow and PyTorch, helping to explain their complementary dominance in the field.

A one-size-fits-all approach does not work well across the spectrum from cloud to tiny edge devices. Different frameworks represent various philosophies around graph execution, declarative versus imperative APIs, and more. Declaratives define what the program should do, while imperatives focus on how it should be done step-by-step. For instance, TensorFlow uses graph execution and declarative-style modeling, while PyTorch adopts eager execution and imperative modeling



TensorFlow	PyTorch
Written in C++ and is, as a result, very fast and efficient.	Written in Python making it more accessible and flattening the learning curve. However, the C++ core means PyTorch is still quite fast.
Feature rich; TensorFlow can be used for training data as well as for inference.	Very flexible; as data size can also be changed during data training.
Very good documentation; TensorFlow has many users and an big community which has led to strong documentation.	Popular at research level; Pytorch was by far the most talked about ML library at CVPR, one of the most important computer vision conferences.
High popularity; TensorFlow has established itself as the most used ML library over a number of years now.	Rapid growth in popularity in both business and research use cases.
Many APIs available; TensorFlow is a library with a rich choice of easy to use APIs.	Many libraries available; PyTorch is composed of multiple libraries and platforms.
Supports JavaScript; TensorFlow supports JavaScript, C++ and Java in addition to Python.	Python-based; PyTorch allows developers to write code in Python
For Mobile & IoT, inferences can be performed with TensorFlow Lite on mobile devices such as Android or iOS, as well as on Edge TPU or Raspberry Pi.	PyTorch API; the PyTorch API is often preferred as it is better designed - plus TensorFlow has historically changed their API frequently.

Figure 6.2: PyTorch vs. TensorFlow: Features and Functions. Source: K&C

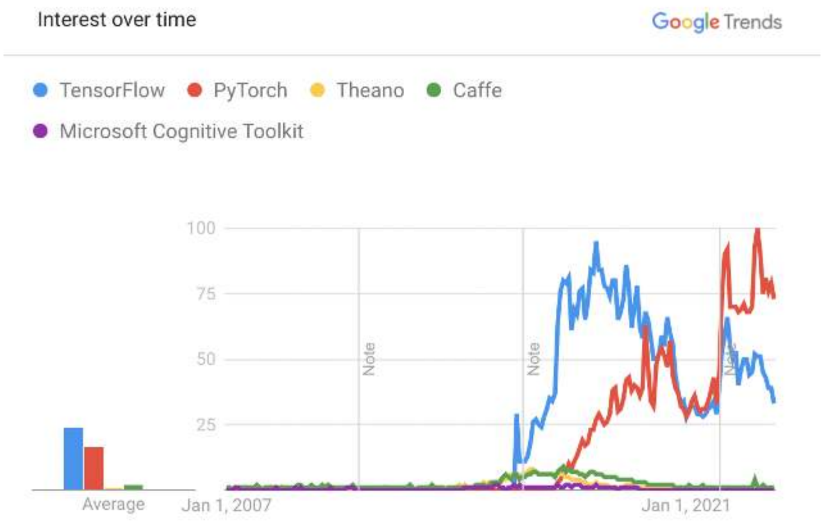


Figure 6.3: Popularity of ML frameworks in the United States as measured by Google web searches. Source: Google.

for more Pythonic flexibility. Each approach carries tradeoffs which we will discuss in Section 6.3.7.

Today's advanced frameworks enable practitioners to develop and deploy increasingly complex models - a key driver of innovation in the AI field. These frameworks continue to evolve and expand their capabilities for the next generation of machine learning. To understand how these systems continue to evolve, we will dive deeper into TensorFlow as an example of how the framework grew in complexity over time.

6.3 Deep Dive into TensorFlow

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 and has since become popular for a wide range of machine learning and deep 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 6.4. 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. In this section, we will mainly discuss only the core package.

6.3.1 TF Ecosystem

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, 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 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.

TensorFlow was developed to address the limitations of DistBelief (Yu et al. 2018)—the framework in use at Google from 2011 to 2015—by providing flexibility along three axes: 1) defining new layers, 2) refining training algorithms, and 3) defining new training algorithms. To understand what limitations in DistBelief led to the development of TensorFlow, we will first give a brief overview of the Parameter Server Architecture that DistBelief employed (Dean et al. 2012).

The Parameter Server (PS) architecture is a popular design for distributing the training of machine learning models, especially deep neural networks, across multiple machines. The fundamental idea is to separate the storage and management of model parameters from the computation used to update these parameters. Typically, parameter

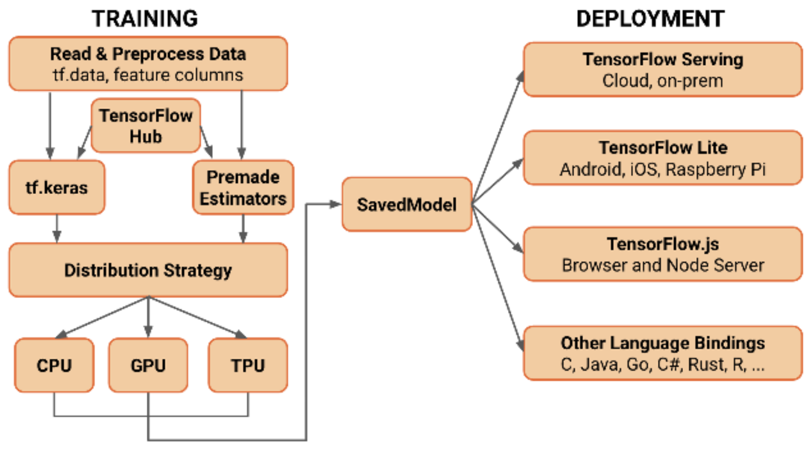


Figure 6.4: Architecture overview of TensorFlow 2.0. Source: Tensorflow.

servers handle the storage and management of model parameters, partitioning them across multiple servers. Worker processes perform the computational tasks, including data processing and computation of gradients, which are then sent back to the parameter servers for updating.

Storage: The stateful parameter server processes handled the storage and management of model parameters. Given the large scale of models and the system's distributed nature, these parameters were shared across multiple parameter servers. Each server maintained a portion of the model parameters, making it "stateful" as it had to maintain and manage this state across the training process.

Computation: The worker processes, which could be run in parallel, were stateless and purely computational. They processed data and computed gradients without maintaining any state or long-term memory (M. Li et al. 2014). Workers did not retain information between different tasks. Instead, they periodically communicated with the parameter servers to retrieve the latest parameters and send back computed gradients.

Caution 9: TensorFlow Core

Let's comprehensively understand core machine learning algorithms using TensorFlow and their practical applications in data analysis and predictive modeling. We will start with linear regression to predict survival rates from the Titanic dataset. Then, using TensorFlow, we will construct classifiers to identify different species of flowers based on their attributes. Next, we will use the K-Means algorithm and its application in segmenting datasets into cohesive clusters. Finally, we will apply hidden Markov models (HMM) to foresee weather patterns.

Open in Colab

Caution 10: TensorFlow Lite

Here, we will see how to build a miniature machine-learning model for microcontrollers. We will build a mini neural network that is streamlined to learn from data even with limited resources and optimized for deployment by shrinking our model for efficient use on microcontrollers. TensorFlow Lite, a powerful technology derived from TensorFlow, shrinks models for tiny devices and helps enable on-device features like image recognition in smart devices. It is used in edge computing to allow for faster analysis and decisions in devices processing data locally.

Open in Colab

DistBelief and its architecture defined above were crucial in enabling distributed deep learning at Google but also introduced limitations that motivated the development of TensorFlow:

6.3.2 Static Computation Graph

Model parameters are distributed across various parameter servers in the parameter server architecture. Since DistBelief was primarily designed for the neural network paradigm, parameters corresponded to a fixed neural network structure. If the computation graph were dynamic, the distribution and coordination of parameters would become significantly more complicated. For example, a change in the graph might require the initialization of new parameters or the removal of existing ones, complicating the management and synchronization tasks of the parameter servers. This made it harder to implement models outside the neural framework or models that required dynamic computation graphs.

TensorFlow was designed as a more general computation framework that expresses computation as a data flow graph. This allows for a wider variety of machine learning models and algorithms outside of neural networks and provides flexibility in refining models.

6.3.3 Usability & Deployment

The parameter server model delineates roles (worker nodes and parameter servers) and is optimized for data center deployments, which might only be optimal for some use cases. For instance, this division introduces overheads or complexities on edge devices or in other non-data center environments.

TensorFlow was built to run on multiple platforms, from mobile devices and edge devices to cloud infrastructure. It also aimed to be lighter and developer-friendly and to provide ease of use between local and distributed training.

6.3.4 Architecture Design

Rather than using the parameter server architecture, TensorFlow deploys tasks across a cluster. These tasks are named processes that can communicate over a network, and each can execute TensorFlow's core construct, the dataflow graph, and interface with various computing devices (like CPUs or GPUs). This graph is a directed representation where nodes symbolize computational operations, and edges depict the tensors (data) flowing between these operations.

Despite the absence of traditional parameter servers, some "PS tasks" still store and manage parameters reminiscent of parameter servers in other systems. The remaining tasks, which usually handle computation, data processing, and gradient calculations, are referred to as "worker tasks." TensorFlow's PS tasks can execute any computation representable by the dataflow graph, meaning they aren't just limited to parameter storage, and the computation can be distributed. This capability makes them significantly more versatile and gives users the power to program the PS tasks using the standard TensorFlow interface, the same one they'd use to define their models. As mentioned above, dataflow graphs' structure also makes them inherently good for parallelism, allowing for the processing of large datasets.

6.3.5 Built-in Functionality & Keras

TensorFlow includes libraries to help users develop and deploy more use-case-specific models, and since this framework is open-source, this list continues to grow. These libraries address the entire ML development lifecycle: data preparation, model building, deployment, and responsible AI.

One of TensorFlow's biggest advantages is its integration with Keras, though, as we will cover in the next section, Pytorch recently added a Keras integration. Keras is another ML framework built to be extremely user-friendly and, as a result, has a high level of abstraction. We will cover Keras in more depth later in this chapter. However, when discussing its integration with TensorFlow, it was important to note that it was originally built to be backend-agnostic. This means users could abstract away these complexities, offering a cleaner, more intuitive way to define and train models without worrying about compatibility issues with different backends. TensorFlow users had some complaints about the usability and readability of TensorFlow's API, so as TF gained prominence, it integrated Keras as its high-level API. This integration offered major benefits to TensorFlow users since it introduced more intuitive readability and portability of models while still taking advantage of powerful backend features, Google support, and infrastructure to deploy models on various platforms.

Caution 11: Exploring Keras: Building, Training, and Evaluating Neural Networks

Here, we'll learn how to use Keras, a high-level neural network API, for model development and training. We will explore the functional API for concise model building, understand loss and metric classes for model evaluation, and use built-in optimizers to update model parameters during training. Additionally, we'll discover how to define custom layers and metrics tailored to our needs. Lastly, we'll look into Keras' training loops to streamline the process of training neural networks on large datasets. This knowledge will empower us to build and optimize neural network models across various machine learning and artificial intelligence applications.

Open in Colab

6.3.6 Limitations and Challenges

TensorFlow is one of the most popular deep learning frameworks, but it has faced criticisms and weaknesses, primarily related to usability and resource usage. While advantageous, the rapid pace of updates through its support from Google has sometimes led to backward compatibility issues, deprecated functions, and shifting documentation. Additionally, even with the Keras implementation, TensorFlow's syntax and learning curve can be difficult for new users. Another major critique of TensorFlow is its high overhead and memory consumption due to the range of built-in libraries and support. While pared-down versions can address some of these concerns, they may still be limited in resource-constrained environments.

6.3.7 PyTorch vs. TensorFlow

PyTorch and TensorFlow have established themselves as frontrunners in the industry. Both frameworks offer robust functionalities but differ in design philosophies, ease of use, ecosystem, and deployment capabilities.

Design Philosophy and Programming Paradigm: PyTorch uses a dynamic computational graph termed eager execution. This makes it intuitive and facilitates debugging since operations are executed immediately and can be inspected on the fly. In comparison, earlier versions of TensorFlow were centered around a static computational graph, which required the graph's complete definition before execution. However, TensorFlow 2.0 introduced eager execution by default, making it more aligned with PyTorch. PyTorch's dynamic nature and Python-based approach have enabled its simplicity and flexibility, particularly for rapid prototyping. TensorFlow's static graph approach in its earlier versions had a steeper learning curve; the introduction of TensorFlow 2.0, with its Keras integration as the high-level API, has significantly simplified the development process.

Deployment: PyTorch is heavily favored in research environments, but deploying PyTorch models in production settings has traditionally been challenging. However, deployment has become more feasible with the introduction of TorchScript, the TorchServe tool, and PyTorch Mobile. TensorFlow stands out for its strong scalability and deployment capabilities, particularly on embedded and mobile platforms with TensorFlow Lite. TensorFlow Serving and TensorFlow.js further facilitate deployment in various environments, thus giving it a broader reach in the ecosystem.

Performance: Both frameworks offer efficient hardware acceleration for their operations. However, TensorFlow has a slightly more robust optimization workflow, such as the XLA (Accelerated Linear Algebra) compiler, which can further boost performance. Its static computational graph was also advantageous for certain optimizations in the early versions.

Ecosystem: PyTorch has a growing ecosystem with tools like TorchServe for serving models and libraries like TorchVision, TorchText, and TorchAudio for specific domains. As we mentioned earlier, TensorFlow has a broad and mature ecosystem. TensorFlow Extended (TFX) provides an end-to-end platform for deploying production machine learning pipelines. Other tools and libraries include TensorFlow Lite, TensorFlow Lite Micro, TensorFlow.js, TensorFlow Hub, and TensorFlow Serving. Table 6.1 provides a comparative analysis:

Table 6.1: Comparison of PyTorch and TensorFlow.

Aspect	Pytorch	TensorFlow
Design Philosophy	Dynamic computational graph (eager execution)	Static computational graph (early versions); Eager execution in TensorFlow 2.0
Deployment	Traditionally challenging; Improved with TorchScript & TorchServe	Scalable, especially on embedded platforms with TensorFlow Lite
Performance & Optimization	Efficient GPU acceleration	Robust optimization with XLA compiler
Ecosystem	TorchServe, TorchVision, TorchText, TorchAudio, PyTorch Mobile	TensorFlow Extended (TFX), TensorFlow Lite, TensorFlow Lite Micro TensorFlow.js, TensorFlow Hub, TensorFlow Serving
Ease of Use	Preferred for its Pythonic approach and rapid prototyping	Initially steep learning curve; Simplified with Keras in TensorFlow 2.0

6.4 Basic Framework Components

Having introduced the popular machine learning frameworks and provided a high-level comparison, this section will introduce you to the core functionalities that form the fabric of these frameworks. It will cover the special structure called tensors, which these frameworks use to handle complex multi-dimensional data more easily. You will also learn how these frameworks represent different types of neural network architectures and their required operations through computational graphs. Additionally, you will see how they offer tools that make the development of machine learning models more abstract and efficient, such as data loaders, implemented loss optimization algorithms, efficient differentiation techniques, and the ability to accelerate your training process on hardware accelerators.

6.4.1 Tensor data structures

As shown in the figure, vectors can be represented as a stack of numbers in a 1-dimensional array. Matrices follow the same idea, and one can think of them as many vectors stacked on each other, making them 2 dimensional. Higher dimensional tensors work the same way. A 3-dimensional tensor, as illustrated in Figure 6.5, is simply a set of matrices stacked on each other in another direction. Therefore, vectors and matrices can be considered special cases of tensors with 1D and 2D dimensions, respectively.

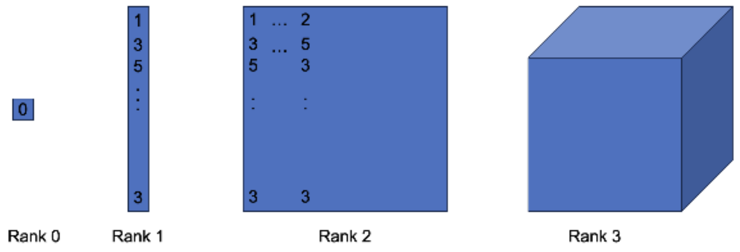


Figure 6.5: Visualization of Tensor Data Structure.

Tensors offer a flexible structure that can represent data in higher dimensions. Figure 6.6 illustrates how this concept applies to image data. As shown in the figure, images are not represented by just one matrix of pixel values. Instead, they typically have three channels, where each channel is a matrix containing pixel values that represent the intensity of red, green, or blue. Together, these channels create a colored image. Without tensors, storing all this information from multiple matrices can be complex. However, as Figure 6.6 illustrates, tensors make it easy to contain image data in a single 3-dimensional structure, with each number representing a certain color value at a specific location in the image.

You don't have to stop there. If we wanted to store a series of images, we could use a 4-dimensional tensor, where the new dimension represents different images. This means you are storing multiple images, each having three matrices that represent the three color channels. This gives you an idea of the usefulness of tensors when dealing with multi-dimensional data efficiently.

Tensors also have a unique attribute that enables frameworks to automatically compute gradients, simplifying the implementation of complex models and optimization algorithms. In machine learning, as dis-

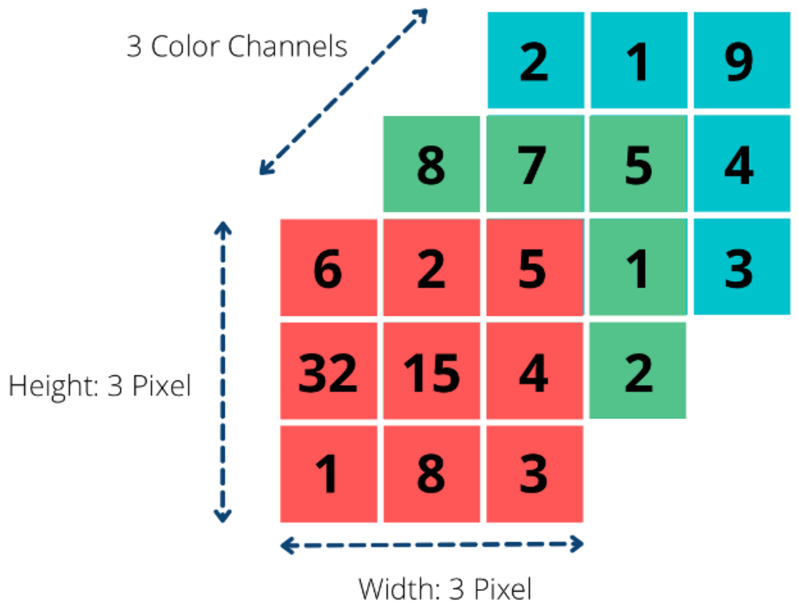


Figure 6.6: Visualization of colored image structure that can be easily stored as a 3D Tensor. Credit: Niklas Lang

cussed in Chapter 3, backpropagation requires taking the derivative of equations. One of the key features of tensors in PyTorch and TensorFlow is their ability to track computations and calculate gradients. This is crucial for backpropagation in neural networks. For example, in PyTorch, you can use the `requires_grad` attribute, which allows you to automatically compute and store gradients during the backward pass, facilitating the optimization process. Similarly, in TensorFlow, `tf.GradientTape` records operations for automatic differentiation.

Consider this simple mathematical equation that you want to differentiate. Mathematically, you can compute the gradient in the following way:

Given:

$$y = x^2$$

The derivative of y with respect to x is:

$$\frac{dy}{dx} = 2x$$

When $x = 2$:

$$\frac{dy}{dx} = 2 * 2 = 4$$

The gradient of y with respect to x, at $x = 2$, is 4.

A powerful feature of tensors in PyTorch and TensorFlow is their ability to easily compute derivatives (gradients). Here are the corresponding code examples in PyTorch and TensorFlow:

6.4.2 PyTorch

```
import torch

# Create a tensor with gradient tracking
x = torch.tensor(2.0, requires_grad=True)

# Define a simple function
y = x ** 2

# Compute the gradient
y.backward()

# Print the gradient
print(x.grad)

# Output
tensor(4.0)
```

6.4.3 TensorFlow

```
import tensorflow as tf

# Create a tensor with gradient tracking
x = tf.Variable(2.0)

# Define a simple function
with tf.GradientTape() as tape:
    y = x ** 2

# Compute the gradient
grad = tape.gradient(y, x)

# Print the gradient
print(grad)
```

```
# Output
tf.Tensor(4.0, shape=(), dtype=float32)
```

This automatic differentiation is a powerful feature of tensors in frameworks like PyTorch and TensorFlow, making it easier to implement and optimize complex machine learning models.

6.4.4 Computational graphs

6.4.4.1 Graph Definition

Computational graphs are a key component of deep learning frameworks like TensorFlow and PyTorch. They allow us to express complex neural network architectures efficiently and differently. A computational graph consists of a directed acyclic graph (DAG) where each node represents an operation or variable, and edges represent data dependencies between them.

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, as depicted in Chapter 3, 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.

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 6.7. The directed acyclic graph computes $z = x \times y$, where each variable is just numbers.

Frameworks like TensorFlow and PyTorch create computational graphs to implement the architectures of neural networks that we typically represent with diagrams. When you define a neural network layer in code (e.g., a dense layer in TensorFlow), the framework constructs a computational graph that includes all the necessary operations (such as matrix multiplication, addition, and activation functions) and their data dependencies. This graph enables the framework to efficiently manage the flow of data, optimize the execution of operations, and automatically compute gradients for training. Underneath the hood, the computational graphs represent abstractions for common layers like convolutional, pooling, recurrent, and dense layers, with data including activations, weights, and biases represented in tensors. This representation allows for efficient compu-

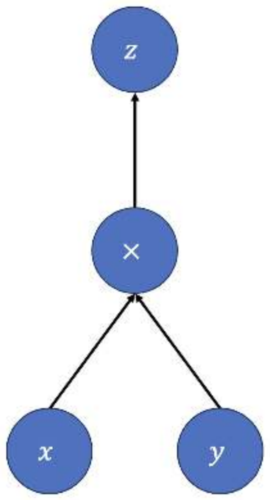


Figure 6.7: Basic example of a computational graph.

tation, leveraging the structure of the graph to parallelize operations and apply optimizations.

Some common layers that computational graphs might implement include convolutional layers, attention layers, recurrent layers, and dense layers. Layers serve as higher-level abstractions that define specific computations on top of the basic operations represented in the graph. For example, a Dense layer performs matrix multiplication and addition between input, weight, and bias tensors. It is important to note that a layer operates on tensors as inputs and outputs; the layer itself is not a tensor. Some key differences between layers and tensors are:

- Layers contain states like weights and biases. Tensors are stateless, just holding data.
- Layers can modify internal state during training. Tensors are immutable/read-only.
- Layers are higher-level abstractions. Tensors are at a lower level and directly represent data and math operations.
- Layers define fixed computation patterns. Tensors flow between layers during execution.
- Layers are used indirectly when building models. Tensors flow between layers during execution.

So, while tensors are a core data structure that layers consume and produce, layers have additional functionality for defining parameterized operations and training. While a layer configures tensor operations under the hood, the layer remains distinct from the tensor objects. The layer abstraction makes building and training neural networks much more intuitive. This abstraction enables developers to build models by stacking these layers together without implementing the layer logic. For example, calling `tf.keras.layers.Conv2D` in TensorFlow creates a convolutional layer. The framework handles computing the convolutions, managing parameters, etc. This simplifies model development, allowing developers to focus on architecture rather than low-level implementations. Layer abstractions use highly optimized implementations for performance. They also enable portability, as the same architecture can run on different hardware backends like GPUs and TPUs.

In addition, computational graphs include activation functions like ReLU, sigmoid, and tanh that are essential to neural networks, and many frameworks provide these as standard abstractions. These functions introduce non-linearities that enable models to approximate complex functions. Frameworks provide these as simple, predefined operations that can be used when constructing models, for example, if.nn.relu in TensorFlow. This abstraction enables flexibility, as developers can easily swap activation functions for tuning performance. Predefined activations are also optimized by the framework for faster execution.

In recent years, models like ResNets and MobileNets have emerged as popular architectures, with current frameworks pre-packaging these as computational graphs. Rather than worrying about the fine details, developers can use them as a starting point, customizing as needed by substituting layers. This simplifies and speeds up model development, avoiding reinventing architectures from scratch. Predefined models include well-tested, optimized implementations that ensure good performance. Their modular design also enables transferring learned features to new tasks via transfer learning. These predefined architectures provide high-performance building blocks to create robust models quickly.

These layer abstractions, activation functions, and predefined architectures the frameworks provide constitute a computational graph. When a user defines a layer in a framework (e.g., `tf.keras.layers.Dense()`), the framework configures computational graph nodes and edges to represent that layer. The layer parameters like weights and biases become variables in the graph. The layer computations become operation nodes (such as the x and y in the figure above). When you call an activation function like `tf.nn.relu()`, the framework adds a ReLU operation node to the graph. Predefined architectures are just pre-configured subgraphs that can be inserted into your model's graph. Thus, model definition via high-level abstractions creates a computational graph—the layers, activations, and architectures we use become graph nodes and edges.

We implicitly construct a computational graph when defining a neural network architecture in a framework. The framework uses this graph to determine operations to run during training and inference. Computational graphs bring several advantages over raw code, and that's one of the core functionalities that is offered by a good ML framework:

- Explicit representation of data flow and operations
- Ability to optimize graph before execution
- Automatic differentiation for training
- Language agnosticism - graph can be translated to run on GPUs, TPUs, etc.
- Portability - graph can be serialized, saved, and restored later

Computational graphs are the fundamental building blocks of ML frameworks. Model definition via high-level abstractions creates a computational graph—the layers, activations, and architectures we use become graph nodes and edges. The framework compilers and optimizers operate on this graph to generate executable code. The abstractions provide a developer-friendly API for building computational graphs. Under the hood, it's still graphs down! So, while you may not directly manipulate graphs as a framework user, they enable your high-level model specifications to be efficiently executed. The abstractions simplify model-building, while computational graphs make it possible.

6.4.4.2 Static vs. Dynamic Graphs

Deep learning frameworks have traditionally followed one of two approaches for expressing computational graphs.

Static graphs (declare-then-execute): With this model, the entire computational graph must be defined upfront before running it. All operations and data dependencies must be specified during the declaration phase. TensorFlow originally followed this static approach -

models were defined in a separate context, and then a session was created to run them. The benefit of static graphs is they allow more aggressive optimization since the framework can see the full graph. However, it also tends to be less flexible for research and interactivity. Changes to the graph require re-declaring the full model.

For example:

```
x = tf.placeholder(tf.float32)
y = tf.matmul(x, weights) + biases
```

In this example, x is a placeholder for input data, and y is the result of a matrix multiplication operation followed by an addition. The model is defined in this declaration phase, where all operations and variables must be specified upfront.

Once the entire graph is defined, the framework compiles and optimizes it. This means that the computational steps are set in stone, and the framework can apply various optimizations to improve efficiency and performance. When you later execute the graph, you provide the actual input tensors, and the pre-defined operations are carried out in the optimized sequence.

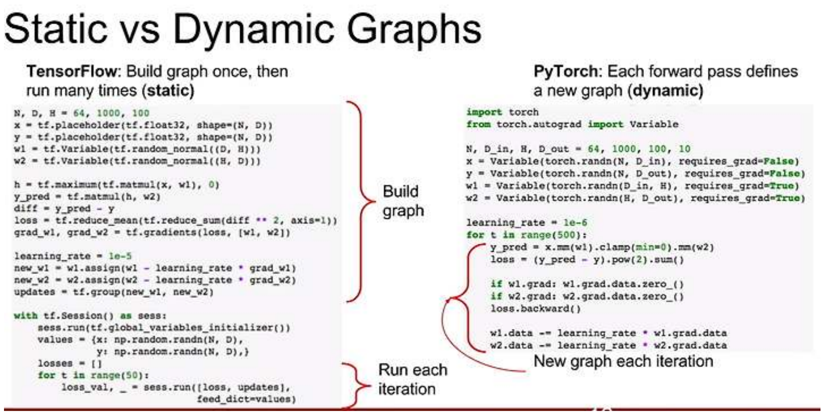
This approach is similar to building a blueprint where every detail is planned before construction begins. While this allows for powerful optimizations, it also means that any changes to the model require redefining the entire graph from scratch.

Dynamic graphs (define-by-run): Unlike declaring (all) first and then executing, the graph is built dynamically as execution happens. There is no separate declaration phase - operations execute immediately as defined. This style is imperative and flexible, facilitating experimentation.

PyTorch uses dynamic graphs, building the graph on the fly as execution happens. For example, consider the following code snippet, where the graph is built as the execution is taking place:

```
x = torch.randn(4,784)
y = torch.matmul(x, weights) + biases
```

The above example does not have separate compile/build/run phases. Ops define and execute immediately. With dynamic graphs, the definition is intertwined with execution, providing a more intuitive, interactive workflow. However, the downside is that there is less potential for optimization since the framework only sees the graph as it is built. Figure 6.8 demonstrates the differences between a static and dynamic computation graph.


Static vs Dynamic Graphs

TensorFlow: Build graph once, then run many times (**static**)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

PyTorch: Each forward pass defines a new graph (**dynamic**)

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Figure 6.8: Comparing static and dynamic graphs. Source: Dev

Recently, the distinction has blurred as frameworks adopt both modes. TensorFlow 2.0 defaults to dynamic graph mode while letting users work with static graphs when needed. Dynamic declaration offers flexibility and ease of use, making frameworks more user-friendly, while static graphs provide optimization benefits at the cost of interactivity. The ideal framework balances these approaches. Table 6.2 compares the pros and cons of static versus dynamic execution graphs:

Table 6.2: Comparison between Static (Declare-then-execute) and Dynamic (Define-by-run) Execution Graphs, highlighting their respective pros and cons.

Execution Graph	Pros	Cons
Static (Declare-then-execute)	• Enable graph optimizations by seeing full model ahead of time • Can export and deploy frozen graphs • Graph is packaged independently of code	• Less flexible for research and iteration • Changes require rebuilding graph • Execution has separate compile and run phases

Execution Graph	Pros	Cons
Dynamic (Define-by-run)	• Intuitive imperative style like Python code • Interleave graph build with execution • Easy to modify graphs • Debugging seamlessly fits workflow	• Harder to optimize without full graph • Possible slowdowns from graph building during execution • Can require more memory

6.4.5 Data Pipeline Tools

Computational graphs can only be as good as the data they learn from and work on. Therefore, feeding training data efficiently is crucial for optimizing deep neural network performance, though it is often overlooked as one of the core functionalities. Many modern AI frameworks provide specialized pipelines to ingest, process, and augment datasets for model training.

6.4.5.1 Data Loaders

At the core of these pipelines are data loaders, which handle reading training examples from sources like files, databases, and object storage. Data loaders facilitate efficient data loading and preprocessing, crucial for deep learning models. For instance, TensorFlow's tf.data dataloading pipeline is designed to manage this process. Depending on the application, deep learning models require diverse data formats such as CSV files or image folders. Some popular formats include:

- **CSV**: A versatile, simple format often used for tabular data.
- **TFRecord**: TensorFlow's proprietary format, optimized for performance.
- **Parquet**: Columnar storage, offering efficient data compression and retrieval.
- **JPEG/PNG**: Commonly used for image data.
- **WAV/MP3**: Prevalent formats for audio data.

Data loaders batch examples to leverage vectorization support in hardware. Batching refers to grouping multiple data points for simultaneous processing, leveraging the vectorized computation capabilities of hardware like GPUs. While typical batch sizes range from 32 to 512 examples, the optimal size often depends on the data's memory footprint and the specific hardware constraints. Advanced loaders can stream virtually unlimited datasets from disk and cloud storage. They stream large datasets from disks or networks instead of fully loading them into memory, enabling unlimited dataset sizes.

Data loaders can also shuffle data across epochs for randomization and preprocess features in parallel with model training to expedite the training process. Randomly shuffling the order of examples between training epochs reduces bias and improves generalization.

Data loaders also support caching and prefetching strategies to optimize data delivery for fast, smooth model training. Caching preprocessed batches in memory allows them to be reused efficiently during multiple training steps and eliminates redundant processing. Prefetching, conversely, involves preloading subsequent batches, ensuring that the model never idles waiting for data.

6.4.6 Data Augmentation

Machine learning frameworks like TensorFlow and PyTorch provide tools to simplify and streamline the process of data augmentation, enhancing the efficiency of expanding datasets synthetically. These frameworks offer integrated functionalities to apply random transformations, such as flipping, cropping, rotating, altering color, and adding noise for images. For audio data, common augmentations involve mixing clips with background noise or modulating speed, pitch, and volume.

By integrating augmentation tools into the data pipeline, frameworks enable these transformations to be applied on the fly during each training epoch. This approach increases the variation in the training data distribution, thereby reducing overfitting and improving model generalization. Figure 6.9 demonstrates the cases of overfitting and underfitting. The use of performant data loaders in combination with extensive augmentation capabilities allows practitioners to efficiently feed massive, varied datasets to neural networks.

These hands-off data pipelines represent a significant improvement in usability and productivity. They allow developers to focus more on model architecture and less on data wrangling when training deep learning models.

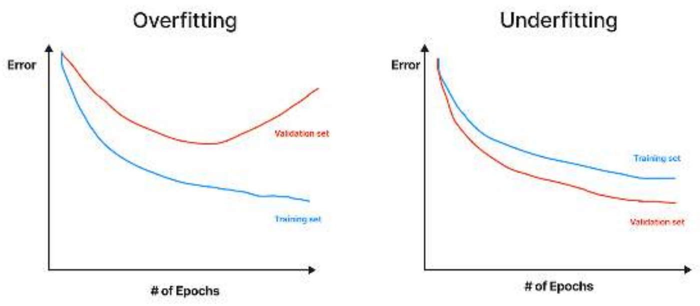


Figure 6.9: Overfitting versus underfitting. Source: Aquarium Learning

6.4.7 Loss Functions and Optimization Algorithms

Training a neural network is fundamentally an iterative process that seeks to minimize a loss function. The goal is to fine-tune the model weights and parameters to produce predictions close to the true target labels. Machine learning frameworks have greatly streamlined this process by offering loss functions and optimization algorithms.

Machine learning frameworks provide implemented loss functions that are needed for quantifying the difference between the model's predictions and the true values. Different datasets require a different loss function to perform properly, as the loss function tells the computer the "objective" for it to aim. Commonly used loss functions include Mean Squared Error (MSE) for regression tasks, Cross-Entropy Loss for classification tasks, and Kullback-Leibler (KL) Divergence for probabilistic models. For instance, TensorFlow's tf.keras.losses holds a suite of these commonly used loss functions.

Optimization algorithms are used to efficiently find the set of model parameters that minimize the loss function, ensuring the model performs well on training data and generalizes to new data. Modern frameworks come equipped with efficient implementations of several optimization algorithms, many of which are variants of gradient descent with stochastic methods and adaptive learning rates. Some examples of these variants are Stochastic Gradient Descent, Adagrad, Adadelta, and Adam. The implementation of such variants are provided in tf.keras.optimizers. More information with clear examples can be found in the AI Training section.

6.4.8 Model Training Support

A compilation step is required before training a defined neural network model. During this step, the neural network's high-level architecture is transformed into an optimized, executable format. This process comprises several steps. The first step is to construct the computational graph, which represents all the mathematical operations and data flow within the model. We discussed this earlier.

During training, the focus is on executing the computational graph. Every parameter within the graph, such as weights and biases, is assigned an initial value. Depending on the chosen initialization method, this value might be random or based on a predefined logic.

The next critical step is memory allocation. Essential memory is reserved for the model's operations on both CPUs and GPUs, ensuring efficient data processing. The model's operations are then mapped to the available hardware resources, particularly GPUs or TPUs, to expedite computation. Once the compilation is finalized, the model is prepared for training.

The training process employs various tools to improve efficiency. Batch processing is commonly used to maximize computational throughput. Techniques like vectorization enable operations on entire data arrays rather than proceeding element-wise, which bolsters speed. Optimizations such as kernel fusion (refer to the Optimizations chapter) amalgamate multiple operations into a single action, minimizing computational overhead. Operations can also be segmented into phases, facilitating the concurrent processing of different mini-batches at various stages.

Frameworks consistently checkpoint the state, preserving intermediate model versions during training. This ensures that progress is recovered if an interruption occurs, and training can be recommenced from the last checkpoint. Additionally, the system vigilantly monitors the model's performance against a validation data set. Should the model begin to overfit (if its performance on the validation set declines), training is automatically halted, conserving computational resources and time.

ML frameworks incorporate a blend of model compilation, enhanced batch processing methods, and utilities such as checkpointing and early stopping. These resources manage the complex aspects of performance, enabling practitioners to zero in on model development and training. As a result, developers experience both speed and ease when utilizing neural networks' capabilities.

6.4.9 Validation and Analysis

After training deep learning models, frameworks provide utilities to evaluate performance and gain insights into the models' workings. These tools enable disciplined experimentation and debugging.

6.4.9.1 Evaluation Metrics

Frameworks include implementations of common evaluation metrics for validation:

- Accuracy - Fraction of correct predictions overall. They are widely used for classification.
- Precision - Of positive predictions, how many were positive. Useful for imbalanced datasets.
- Recall - Of actual positives, how many did we predict correctly? Measures completeness.
- F1-score - Harmonic mean of precision and recall. Combines both metrics.
- AUC-ROC - Area under ROC curve. They are used for classification threshold analysis.
- MAP - Mean Average Precision. Evaluate ranked predictions in retrieval/detection.
- Confusion Matrix - Matrix that shows the true positives, true negatives, false positives, and false negatives. Provides a more detailed view of classification performance.

These metrics quantify model performance on validation data for comparison.

6.4.9.2 Visualization

Visualization tools provide insight into models:

- Loss curves - Plot training and validation loss over time to spot Overfitting.
- Activation grids - Illustrate features learned by convolutional filters.
- Projection - Reduce dimensionality for intuitive visualization.

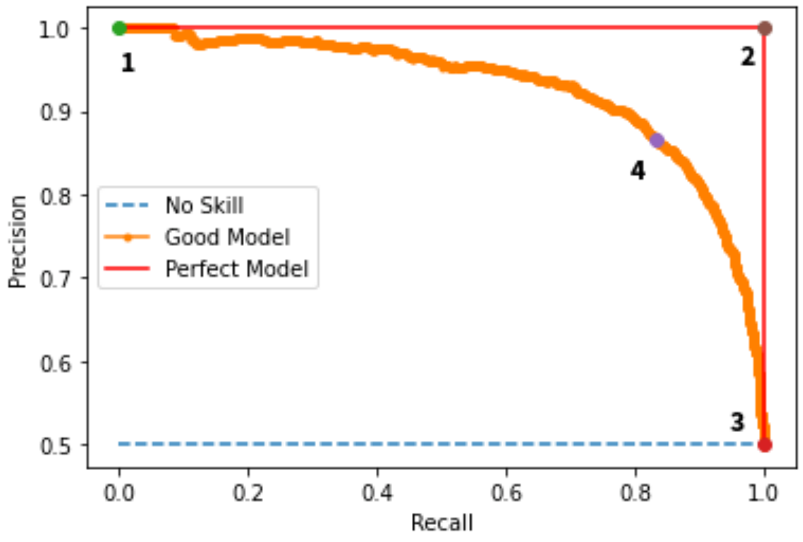


Figure 6.10: Reading a precision-recall curve. Source: AIM

- Precision-recall curves - Assess classification tradeoffs. Figure 6.10 shows an example of a precision-recall curve.

Tools like TensorBoard for TensorFlow and TensorWatch for PyTorch enable real-time metrics and visualization during training.

6.4.10 Differentiable programming

Machine learning training methods such as backpropagation rely on the change in the loss function with respect to the change in weights (which essentially is the definition of derivatives). Thus, the ability to quickly and efficiently train large machine learning models relies on the computer's ability to take derivatives. This makes differentiable programming one of the most important elements of a machine learning framework.

We can use four primary methods to make computers take derivatives. First, we can manually figure out the derivatives by hand and input them into the computer. This would quickly become a nightmare with many layers of neural networks if we had to compute all the derivatives in the backpropagation steps by hand. Another method is symbolic differentiation using computer algebra systems such as Mathematica, which can introduce a layer of inefficiency, as there needs to be a level of abstraction to take derivatives. Numerical derivatives, the practice of approximating gradients using finite difference methods, suffer from many problems, including high computational costs and larger grid sizes, leading to many errors. This leads to automatic differentiation, which exploits the primitive functions that computers use to represent operations to obtain an exact derivative. With automatic differentiation, the computational complexity of computing the gradient is proportional to computing the function itself. Intricacies of automatic differentiation are not dealt with by end users now, but resources to learn more can be found widely, such as from here. Today's automatic differentiation and differentiable programming are ubiquitous and are done efficiently and automatically by modern machine learning frameworks.

6.4.11 Hardware Acceleration

The trend to continuously train and deploy larger machine-learning models has made hardware acceleration support necessary for machine-learning platforms. Figure 6.11 shows the large number of companies that are offering hardware accelerators in different domains, such as "Very Low Power" and "Embedded" machine learning. Deep layers of neural networks require many matrix multiplications, which attract hardware that can compute matrix operations quickly and in parallel. In this landscape, two hardware architectures, the GPU and TPU, have emerged as leading choices for training machine learning models.

The use of hardware accelerators began with AlexNet, which paved the way for future works to use GPUs as hardware accelerators for training computer vision models. GPUs, or Graphics Processing Units, excel in handling many computations at once, making them ideal for the matrix operations central to neural network training. Their architecture, designed for rendering graphics, is perfect for the mathematical operations required in machine learning. While they are very useful for machine learning tasks and have been implemented in many hardware platforms, GPUs are still general purpose in that they can be used for other applications.

On the other hand, Tensor Processing Units (TPU) are hardware units designed specifically for neural networks. They focus on the multiply and accumulate (MAC) operation, and their hardware consists of a large hardware matrix that contains elements that efficiently compute the MAC operation. This concept, called the systolic array architecture, was pioneered by Kung and Leiserson (1979), but has proven to be a useful structure to efficiently compute matrix products and other operations within neural networks (such as convolutions).

While TPUs can drastically reduce training times, they also have disadvantages. For example, many operations within the machine learning frameworks (primarily TensorFlow here since the TPU directly integrates with it) are not supported by TPUs. They cannot also support custom operations from the machine learning frameworks, and the network design must closely align with the hardware capabilities.

Today, NVIDIA GPUs dominate training, aided by software libraries like CUDA, cuDNN, and TensorRT. Frameworks also include optimizations to maximize performance on these hardware types, such as pruning unimportant connections and fusing layers. Combining these techniques with hardware acceleration provides greater efficiency. For inference, hardware is increasingly moving towards optimized ASICs and SoCs. Google's TPUs accelerate models in data centers, while Apple, Qualcomm, the NVIDIA Jetson family, and others now produce AI-focused mobile chips.

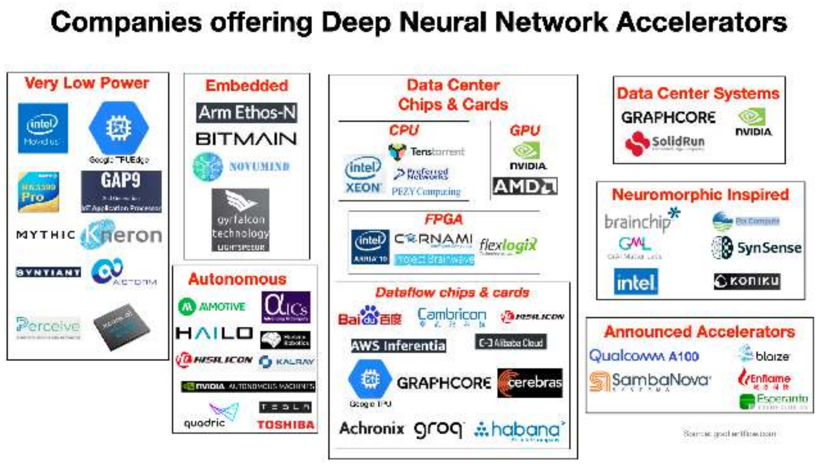


Figure 6.11: Companies offering ML hardware accelerators. Source: Gradient Flow.

6.5 Advanced Features

Beyond providing the essential tools for training machine learning models, frameworks also offer advanced features. These features include distributing training across different hardware platforms, fine-tuning large pre-trained models with ease, and facilitating federated learning. Implementing these capabilities independently would be highly complex and resource-intensive, but frameworks simplify these processes, making advanced machine learning techniques more accessible.

6.5.1 Distributed training

As machine learning models have become larger over the years, it has become essential for large models to use multiple computing nodes in the training process. This process, distributed learning, has allowed for higher training capabilities but has also imposed challenges in implementation.

We can consider three different ways to spread the work of training machine learning models to multiple computing nodes. Input data partitioning (or data parallelism) refers to multiple processors running the same model on different input partitions. This is the easiest implementation and is available for many machine learning frameworks. The more challenging distribution of work comes with model parallelism, which refers to multiple computing nodes working on different parts of the model, and pipelined model parallelism, which refers to multiple computing nodes working on different layers of the model on the same input. The latter two mentioned here are active research areas.

ML frameworks that support distributed learning include TensorFlow (through its tf.distribute module), PyTorch (through its torch.nn.DataParallel and torch.nn.DistributedDataParallel modules), and MXNet (through its gluon API).

6.5.2 Model Conversion

Machine learning models have various methods to be represented and used within different frameworks and for different device types. For example, a model can be converted to be compatible with inference frameworks within the mobile device. The default format for TensorFlow models is checkpoint files containing weights and architectures, which are needed to retrain the models. However, models are typically converted to TensorFlow Lite format for mobile deployment. TensorFlow Lite uses a compact flat buffer representation and optimizations for fast inference on mobile hardware, discarding all the unnecessary baggage associated with training metadata, such as checkpoint file structures.

Model optimizations like quantization (see Optimizations chapter) can further optimize models for target architectures like mobile. This reduces the precision of weights and activations to `uint8` or `int8` for a smaller footprint and faster execution with supported hardware accelerators. For post-training quantization, TensorFlow's converter handles analysis and conversion automatically.

Frameworks like TensorFlow simplify deploying trained models to mobile and embedded IoT devices through easy conversion APIs for TFLite format and quantization. Ready-to-use conversion enables high-performance inference on mobile without a manual optimization burden. Besides TFLite, other common targets include TensorFlow.js for web deployment, TensorFlow Serving for cloud services, and TensorFlow Hub for transfer learning. TensorFlow's conversion utilities handle these scenarios to streamline end-to-end workflows.

More information about model conversion in TensorFlow is linked here.

6.5.3 AutoML, No-Code/Low-Code ML

In many cases, machine learning can have a relatively high barrier of entry compared to other fields. To successfully train and deploy models, one needs to have a critical understanding of a variety of disciplines, from data science (data processing, data cleaning), model structures (hyperparameter tuning, neural network architecture), hardware (acceleration, parallel processing), and more depending on the problem at hand. The complexity of these problems has led to the introduction of frameworks such as AutoML, which tries to make "Machine learning available for non-Machine Learning experts" and to "automate research in machine learning." They have constructed AutoWEKA, which aids in the complex process of hyperparameter selection, and Auto-sklearn and Auto-pytorch, an extension of AutoWEKA into the popular sklearn and PyTorch Libraries.

While these efforts to automate parts of machine learning tasks are underway, others have focused on making machine learning models easier by deploying no-code/low-code machine learning, utilizing a drag-and-drop interface with an easy-to-navigate user interface. Companies such as Apple, Google, and Amazon have already created these easy-to-use platforms to allow users to construct machine learning models that can integrate into their ecosystem.

These steps to remove barriers to entry continue to democratize machine learning, make it easier for beginners to access, and simplify workflow for experts.

6.5.4 Advanced Learning Methods

6.5.4.1 Transfer Learning

Transfer learning is the practice of using knowledge gained from a pretrained model to train and improve the performance of a model for a different task. For example, models such as MobileNet and ResNet are trained on the ImageNet dataset. To do so, one may freeze the pre-trained model, utilizing it as a feature extractor to train a much smaller model built on top of the feature extraction. One can also fine-tune the entire model to fit the new task. Machine learning frameworks make it easy to load pre-trained models, freeze specific layers, and train custom layers on top. They simplify this process by providing intuitive APIs and easy access to large repositories of pre-trained models.

Transfer learning, while powerful, comes with challenges. One significant issue is the modified model's potential inability to conduct its original tasks after transfer learning. To address these challenges, researchers have proposed various solutions. For example, Zhizhong Li and Hoiem (2018) introduced the concept of "Learning without Forgetting" in their paper "Learning without Forgetting", which has since been implemented in modern machine learning platforms. Figure 6.12 provides a simplified illustration of the transfer learning concept:

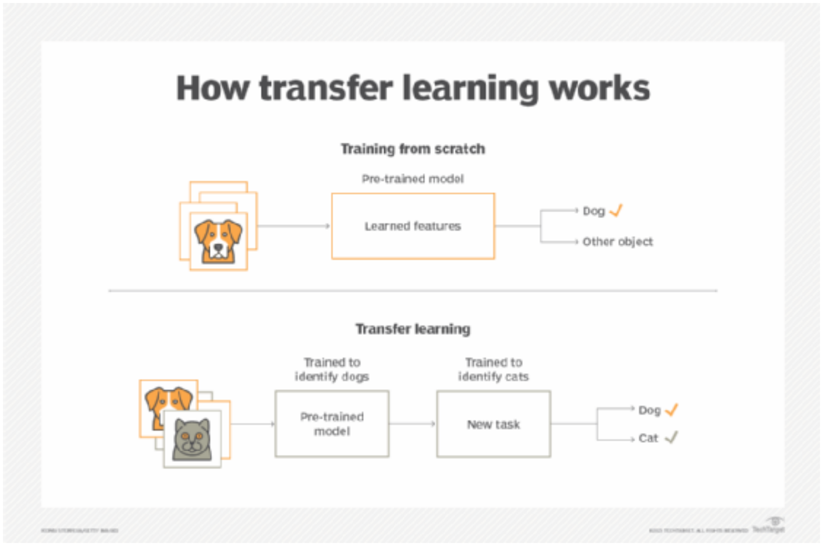


Figure 6.12: Transfer learning. Source: Tech Target

As shown in Figure 6.12, transfer learning involves taking a model trained on one task (the source task) and adapting it to perform a new, related task (the target task). This process allows the model to leverage knowledge gained from the source task, potentially improving performance and reducing training time on the target task. However, as mentioned earlier, care must be taken to ensure that the model doesn't "forget" its ability to perform the original task during this process.

6.5.4.2 Federated Learning

Federated learning by McMahan et al. (2017a) is a form of distributed computing that involves training models on personal devices rather than centralizing the data on a single server (Figure 12.7). Initially, a base global model is trained on a central server to be distributed to all devices. Using this base model, the devices individually compute the gradients and send them back to the central hub. Intuitively, this transfers model parameters instead of the data itself. Federated learning enhances privacy by keeping sensitive data on local devices and only sharing model updates with a central server. This method is particularly useful when dealing with sensitive data or when a large-scale infrastructure is impractical.

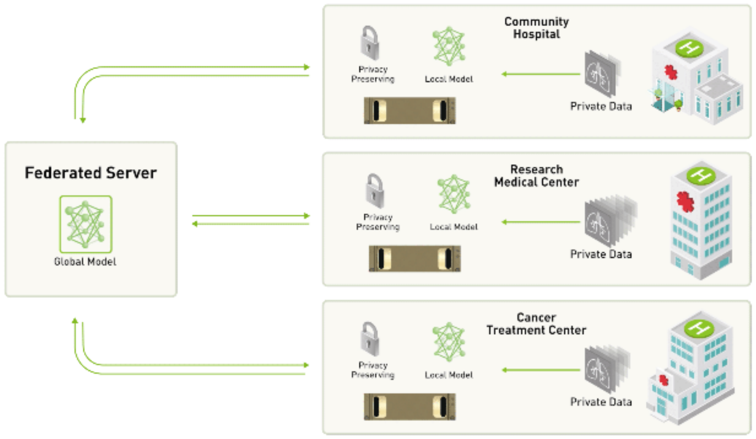


Figure 6.13: A centralized-server approach to federated learning. Source: NVIDIA.

However, federated learning faces challenges such as ensuring data accuracy, managing non-IID (independent and identically distributed) data, dealing with unbalanced data production, and overcoming communication overhead and device heterogeneity. Privacy and security concerns, such as gradient inversion attacks, also pose significant challenges.

Machine learning frameworks simplify the implementation of federated learning by providing necessary tools and libraries. For example, TensorFlow Federated (TFF) offers an open-source framework to support federated learning. TFF allows developers to simulate and implement federated learning algorithms, offering a federated core for low-level operations and high-level APIs for common federated tasks. It seamlessly integrates with TensorFlow, enabling the use of TensorFlow models and optimizers in a federated setting. TFF supports secure aggregation techniques to improve privacy and allows for customization of federated learning algorithms. By leveraging these tools, developers can efficiently distribute training, fine-tune pre-trained models, and handle federated learning's inherent complexities.

Other open source programs such as Flower have also been developed to simplify implementing federated learning with various machine learning frameworks.

6.6 Framework Specialization

Thus far, we have talked about ML frameworks generally. However, typically, frameworks are optimized based on the target environment's computational capabilities and application requirements, ranging from the cloud to the edge to tiny devices. Choosing the right framework is crucial based on the target environment for deployment. This section provides an overview of the major types of AI frameworks tailored for cloud, edge, and TinyML environments to help understand the similarities and differences between these ecosystems.

6.6.1 Cloud

Cloud-based AI frameworks assume access to ample computational power, memory, and storage resources in the cloud. They generally support both training and inference. Cloud-based AI frameworks are suited for applications where data can be sent to the cloud for processing, such as cloud-based AI services, large-scale data analytics, and web applications. Popular cloud AI frameworks include the ones we mentioned earlier, such as TensorFlow, PyTorch, MXNet, Keras, etc. These frameworks utilize GPUs, TPUs, distributed training, and AutoML to deliver scalable AI. Concepts like model serving, MLOps, and AIOps relate to the operationalization of AI in the cloud. Cloud AI powers services like Google Cloud AI and enables transfer learning using pre-trained models.

6.6.2 Edge

Edge AI frameworks are tailored to deploy AI models on IoT devices, smartphones, and edge servers. Edge AI frameworks are optimized for devices with moderate computational resources, balancing power and performance. Edge AI frameworks are ideal for applications requiring real-time or near-real-time processing, including robotics, autonomous vehicles, and smart devices. Key edge AI frameworks include TensorFlow Lite, PyTorch Mobile, CoreML, and others. They employ optimizations like model compression, quantization, and efficient neural network architectures. Hardware support includes CPUs, GPUs, NPUs, and accelerators like the Edge TPU. Edge AI enables use cases like mobile vision, speech recognition, and real-time anomaly detection.

6.6.3 Embedded

TinyML frameworks are specialized for deploying AI models on extremely resource-constrained devices, specifically microcontrollers and sensors within the IoT ecosystem. TinyML frameworks are designed for devices with limited resources, emphasizing minimal memory and power consumption. TinyML frameworks are specialized for use cases on resource-constrained IoT devices for predictive maintenance, gesture recognition, and environmental monitoring applications. Major TinyML frameworks include TensorFlow Lite Micro, uTensor, and ARM NN. They optimize complex models to fit within kilobytes of memory through techniques like quantization-aware training and reduced precision. TinyML allows intelligent sensing across battery-powered devices, enabling collaborative learning via federated learning. The choice of framework involves balancing model performance and computational constraints of the target platform, whether cloud, edge, or TinyML. Table 6.3 compares the major AI frameworks across cloud, edge, and TinyML environments:

Table 6.3: Comparison of framework types for Cloud AI, Edge AI, and TinyML.

Framework Type	Examples	Key Technologies	Use Cases
Cloud AI	TensorFlow, PyTorch, MXNet, Keras	GPUs, TPUs, distributed training, AutoML, MLOps	Cloud services, web apps, big data analytics
Edge AI	TensorFlow Lite, PyTorch Mobile, Core ML	Model optimization, compression, quantization, efficient NN architectures	Mobile apps, autonomous systems, real-time processing

Framework Type	Examples	Key Technologies	Use Cases
TinyML	TensorFlow Lite Micro, uTensor, ARM NN	Quantization-aware training, reduced precision, neural architecture search	IoT sensors, wearables, predictive maintenance, gesture recognition

Key differences:

- Cloud AI leverages massive computational power for complex models using GPUs/TPUs and distributed training
- Edge AI optimizes models to run locally on resource-constrained edge devices.
- TinyML fits models into extremely low memory and computes environments like microcontrollers

6.7 Embedded AI Frameworks

6.7.1 Resource Constraints

Embedded systems face severe resource constraints that pose unique challenges when deploying machine learning models compared to traditional computing platforms. For example, microcontroller units (MCUs) commonly used in IoT devices often have:

- **RAM** ranges from tens of kilobytes to a few megabytes. The popular ESP8266 MCU has around 80KB RAM available to developers. This contrasts with 8GB or more on typical laptops and desktops today.
- **Flash storage** ranges from hundreds of kilobytes to a few megabytes. The Arduino Uno microcontroller provides just 32KB of code storage. Standard computers today have disk storage in the order of terabytes.
- **Processing power** from just a few MHz to approximately 200MHz. The ESP8266 operates at 80MHz. This is several orders of magnitude slower than multi-GHz multi-core CPUs in servers and high-end laptops.

These tight constraints often make training machine learning models directly on microcontrollers infeasible. The limited RAM precludes handling large datasets for training. Energy usage for training would also quickly deplete battery-powered devices. Instead, models are trained on resource-rich systems and deployed on microcontrollers for optimized inference. But even inference poses challenges:

1. **Model Size:** AI models are too large to fit on embedded and IoT devices. This necessitates model compression techniques, such as quantization, pruning, and knowledge distillation. Additionally, as we will see, many of the frameworks used by developers for AI development have large amounts of overhead and built-in libraries that embedded systems can't support.

2. **Complexity of Tasks:** With only tens of KBs to a few MBs of RAM, IoT devices and embedded systems are constrained in the complexity of tasks they can handle. Tasks that require large datasets or sophisticated algorithms—for example, LLMs—that would run smoothly on traditional computing platforms might be infeasible on embedded systems without compression or other optimization techniques due to memory limitations.

3. **Data Storage and Processing:** Embedded systems often process data in real time and might only store small amounts locally. Conversely, traditional computing systems can hold and process large datasets in memory, enabling faster data operations analysis and real-time updates.

4. **Security and Privacy:** Limited memory also restricts the complexity of security algorithms and protocols, data encryption, reverse engineering protections, and more that can be implemented on the device. This could make some IoT devices more vulnerable to attacks.

Consequently, specialized software optimizations and ML frameworks tailored for microcontrollers must work within these tight resource bounds. Clever optimization techniques like quantization, pruning, and knowledge distillation compress models to fit within limited memory (see Optimizations section). Learnings from neural architecture search help guide model designs.

Hardware improvements like dedicated ML accelerators on microcontrollers also help alleviate constraints. For instance, Qualcomm's Hexagon DSP accelerates TensorFlow Lite models on Snapdragon mobile chips. Google's Edge TPU packs ML performance into a tiny ASIC for edge devices. ARM Ethos-U55 offers efficient inference on

Cortex-M class microcontrollers. These customized ML chips unlock advanced capabilities for resource-constrained applications.

Due to limited processing power, it's almost always infeasible to train AI models on IoT or embedded systems. Instead, models are trained on powerful traditional computers (often with GPUs) and then deployed on the embedded device for inference. TinyML specifically deals with this, ensuring models are lightweight enough for real-time inference on these constrained devices.

6.7.2 Frameworks & Libraries

Embedded AI frameworks are software tools and libraries designed to enable AI and ML capabilities on embedded systems. These frameworks are essential for bringing AI to IoT devices, robotics, and other edge computing platforms, and they are designed to work where computational resources, memory, and power consumption are limited.

6.7.3 Challenges

While embedded systems present an enormous opportunity for deploying machine learning to enable intelligent capabilities at the edge, these resource-constrained environments pose significant challenges. Unlike typical cloud or desktop environments rich with computational resources, embedded devices introduce severe constraints around memory, processing power, energy efficiency, and specialized hardware. As a result, existing machine learning techniques and frameworks designed for server clusters with abundant resources do not directly translate to embedded systems. This section uncovers some of the challenges and opportunities for embedded systems and ML frameworks.

6.7.3.1 Fragmented Ecosystem

The lack of a unified ML framework led to a highly fragmented ecosystem. Engineers at companies like STMicroelectronics, NXP Semiconductors, and Renesas had to develop custom solutions tailored to their specific microcontroller and DSP architectures. These ad-hoc frameworks required extensive manual optimization for each low-level hardware platform. This made porting models extremely difficult, requiring redevelopment for new Arm, RISC-V, or proprietary architectures.

6.7.3.2 Disparate Hardware Needs

Without a shared framework, there was no standard way to assess hardware's capabilities. Vendors like Intel, Qualcomm, and NVIDIA created integrated solutions, blending models and improving software and hardware. This made it hard to discern the sources of performance gains - whether new chip designs like Intel's low-power x86 cores or software optimizations were responsible. A standard framework was needed so vendors could evaluate their hardware's capabilities fairly and reproducibly.

6.7.3.3 Lack of Portability

With standardized tools, adapting models trained in common frameworks like TensorFlow or PyTorch to run efficiently on microcontrollers was easier. It required time-consuming manual translation of models to run on specialized DSPs from companies like CEVA or low-power Arm M-series cores. No turnkey tools were enabling portable deployment across different architectures.

6.7.3.4 Incomplete Infrastructure

The infrastructure to support key model development workflows needed to be improved. More support is needed for compression techniques to fit large models within constrained memory budgets. Tools for quantization to lower precision for faster inference were missing. Standardized APIs for integration into applications were incomplete. Essential functionality like on-device debugging, metrics, and performance profiling was absent. These gaps increased the cost and difficulty of embedded ML development.

6.7.3.5 No Standard Benchmark

Without unified benchmarks, there was no standard way to assess and compare the capabilities of different hardware platforms from vendors like NVIDIA, Arm, and Ambiq Micro. Existing evaluations relied on proprietary benchmarks tailored to showcase the strengths of particular chips. This made it impossible to measure hardware improvements objectively in a fair, neutral manner. The Benchmarking AI chapter discusses this topic in more detail.

6.7.3.6 Minimal Real-World Testing

Much of the benchmarks relied on synthetic data. Rigorously testing models on real-world embedded applications was difficult without standardized datasets and benchmarks, raising questions about how performance claims would translate to real-world usage. More extensive testing was needed to validate chips in actual use cases.

The lack of shared frameworks and infrastructure slowed TinyML adoption, hampering the integration of ML into embedded products. Recent standardized frameworks have begun addressing these issues through improved portability, performance profiling, and benchmarking support. However, ongoing innovation is still needed to enable seamless, cost-effective deployment of AI to edge devices.

6.7.3.7 Summary

The absence of standardized frameworks, benchmarks, and infrastructure for embedded ML has traditionally hampered adoption. However, recent progress has been made in developing shared frameworks like TensorFlow Lite Micro and benchmark suites like MLPerf Tiny that aim to accelerate the proliferation of TinyML solutions. However, overcoming the fragmentation and difficulty of embedded deployment remains an ongoing process.

6.8 Examples

Machine learning deployment on microcontrollers and other embedded devices often requires specially optimized software libraries and frameworks to work within tight memory, compute, and power constraints. Several options exist for performing inference on such resource-limited hardware, each with its approach to optimizing model execution. This section will explore the key characteristics and design principles behind TFLite Micro, TinyEngine, and CMSIS-NN, providing insight into how each framework tackles the complex problem of high-accuracy yet efficient neural network execution on microcontrollers. It will also showcase different approaches for implementing efficient TinyML frameworks.

Table 6.4 summarizes the key differences and similarities between these three specialized machine-learning inference frameworks for embedded systems and microcontrollers.

Table 6.4: Comparison of frameworks: TensorFlow Lite Micro, TinyEngine, and CMSIS-NN

Framework	TensorFlow Lite Micro	TinyEngine	CMSIS-NN
Approach	Interpreter-based	Static compilation	Optimized neural network kernels
Hardware Focus	General embedded devices	Microcontrollers	ARM Cortex-M processors
Arithmetic Support	Floating point	Floating point, fixed point	Floating point, fixed point
Model Support	General neural network models	Models co-designed with TinyNAS	Common neural network layer types
Code Footprint	Larger due to inclusion of interpreter and ops	Small, includes only ops needed for model	Lightweight by design
Latency	Higher due to interpretation overhead	Very low due to compiled model	Low latency focus
Memory Management	Dynamically managed by interpreter	Model-level optimization	Tools for efficient allocation
Optimization Approach	Some code generation features	Specialized kernels, operator fusion	Architecture-specific assembly optimizations
Key Benefits	Flexibility, portability, ease of updating models	Maximizes performance, optimized memory usage	Hardware acceleration, standardized API, portability

We will understand each of these in greater detail in the following sections.

6.8.1 Interpreter

TensorFlow Lite Micro (TFLM) is a machine learning inference framework designed for embedded devices with limited resources. It uses an interpreter to load and execute machine learning models, which provides flexibility and ease of updating models in the field (David et al. 2021).

Traditional interpreters often have significant branching overhead, which can reduce performance. However, machine learning model interpretation benefits from the efficiency of long-running kernels, where each kernel runtime is relatively large and helps mitigate interpreter overhead.

An alternative to an interpreter-based inference engine is to generate native code from a model during export. This can improve performance, but it sacrifices portability and flexibility, as the generated code needs recompilation for each target platform and must be replaced entirely to modify a model.

TFLM balances the simplicity of code compilation and the flexibility of an interpreter-based approach by incorporating certain code-generation features. For example, the library can be constructed solely from source files, offering much of the compilation simplicity associated with code generation while retaining the benefits of an interpreter-based model execution framework.

An interpreter-based approach offers several benefits over code generation for machine learning inference on embedded devices:

- **Flexibility:** Models can be updated in the field without recompiling the entire application.
- **Portability:** The interpreter can be used to execute models on different target platforms without porting the code.
- **Memory efficiency:** The interpreter can share code across multiple models, reducing memory usage.
- **Ease of development:** Interpreters are easier to develop and maintain than code generators.

TensorFlow Lite Micro is a powerful and flexible framework for machine learning inference on embedded devices. Its interpreter-based approach offers several benefits over code generation, including flexibility, portability, memory efficiency, and ease of development.

6.8.2 Compiler-based

TinyEngine is an ML inference framework designed specifically for resource-constrained microcontrollers. It employs several optimizations to enable high-accuracy neural network execution within the tight constraints of memory, computing, and storage on microcontrollers (J. Lin et al. 2020).

While inference frameworks like TFLite Micro use interpreters to execute the neural network graph dynamically at runtime, this adds significant overhead regarding memory usage to store metadata, interpretation latency, and lack of optimizations. However, TFLite argues that the overhead is small. TinyEngine eliminates this overhead by employing a code generation approach. It analyzes the network graph during compilation and generates specialized code to execute just that model. This code is natively compiled into the application binary, avoiding runtime interpretation costs.

Conventional ML frameworks schedule memory per layer, trying to minimize usage for each layer separately. TinyEngine does model-level scheduling instead of analyzing memory usage across layers. It allocates a common buffer size based on the maximum memory needs of all layers. This buffer is then shared efficiently across layers to increase data reuse.

TinyEngine also specializes in the kernels for each layer through techniques like tiling, unrolling, and fusing operators. For example, it will generate unrolled compute kernels with the number of loops needed for a 3x3 or 5x5 convolution. These specialized kernels extract maximum performance from the microcontroller hardware. It uses optimized depthwise convolutions to minimize memory allocations by computing each channel's output in place over the input channel data. This technique exploits the channel-separable nature of depthwise convolutions to reduce peak memory size.

Like TFLite Micro, the compiled TinyEngine binary only includes operations needed for a specific model rather than all possible operations. This results in a very small binary footprint, keeping code size low for memory-constrained devices.

One difference between TFLite Micro and TinyEngine is that the latter is co-designed with "TinyNAS," an architecture search method for microcontroller models similar to differential NAS for microcontrollers. TinyEngine's efficiency allows for exploring larger and more accurate models through NAS. It also provides feedback to TinyNAS on which models can fit within the hardware constraints.

Through various custom techniques, such as static compilation, model-based scheduling, specialized kernels, and co-design with NAS, TinyEngine enables high-accuracy deep learning inference within microcontrollers' tight resource constraints.

6.8.3 Library

CMSIS-NN, standing for Cortex Microcontroller Software Interface Standard for Neural Networks, is a software library devised by ARM. It offers a standardized interface for deploying neural network inference on microcontrollers and embedded systems, focusing on optimization for ARM Cortex-M processors (Lai, Suda, and Chandra 2018a).

Neural Network Kernels: CMSIS-NN has highly efficient kernels that handle fundamental neural network operations such as convolution, pooling, fully connected layers, and activation functions. It caters to a broad range of neural network models by supporting floating and fixed-point arithmetic. The latter is especially beneficial for resource-constrained devices as it curtails memory and computational requirements (Quantization).

Hardware Acceleration: CMSIS-NN harnesses the power of Single Instruction, Multiple Data (SIMD) instructions available on many Cortex-M processors. This allows for parallel processing of multiple data elements within a single instruction, thereby boosting computational efficiency. Certain Cortex-M processors feature Digital Signal Processing (DSP) extensions that CMSIS-NN can exploit for accelerated neural network execution. The library also incorporates assembly-level optimizations tailored to specific microcontroller architectures to improve performance further.

Standardized API: CMSIS-NN offers a consistent and abstracted API that protects developers from the complexities of low-level hardware details. This makes the integration of neural network models into applications simpler. It may also encompass tools or utilities for converting popular neural network model formats into a format that is compatible with CMSIS-NN.

Memory Management: CMSIS-NN provides functions for efficient memory allocation and management, which is vital in embedded systems where memory resources are scarce. It ensures optimal memory usage during inference and, in some instances, allows in-place operations to decrease memory overhead.

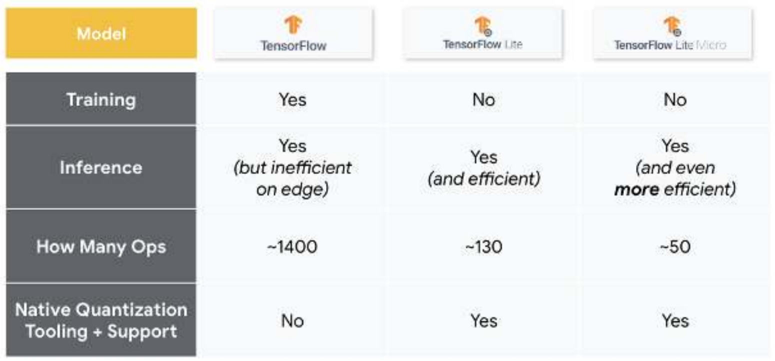
Portability: CMSIS-NN is designed for portability across various Cortex-M processors. This enables developers to write code that can operate on different microcontrollers without significant modifications.

Low Latency: CMSIS-NN minimizes inference latency, making it an ideal choice for real-time applications where swift decision-making is paramount.

Energy Efficiency: The library is designed with a focus on energy efficiency, making it suitable for battery-powered and energy-constrained devices.

6.9 Choosing the Right Framework

Choosing the right machine learning framework for a given application requires carefully evaluating models, hardware, and software considerations. Figure 6.14 provides a comparison of different TensorFlow frameworks, which we'll discuss in more detail:



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

Figure 6.14: TensorFlow Framework Comparison - General. Source: TensorFlow.

Analyzing these three aspects—models, hardware, and software—as depicted in Figure 6.14, ML engineers can select the optimal framework and customize it as needed for efficient and performant on-device ML applications. The goal is to balance model complexity, hardware limitations, and software integration to design a tailored ML pipeline for embedded and edge devices. As we examine the differences shown in Figure 6.14, we'll gain insights into how to pick the right framework and understand what causes the variations between frameworks.

6.9.1 Model

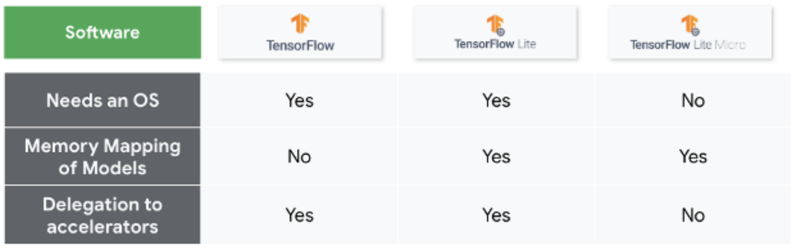
Figure 6.14 illustrates the key differences between TensorFlow variants, particularly in terms of supported operations (ops) and features. TensorFlow supports significantly more operations than TensorFlow Lite and TensorFlow Lite Micro, as it is typically used for research or cloud deployment, which require a large number of and more flexibility with operators.

The figure clearly demonstrates this difference in op support across the frameworks. TensorFlow Lite supports select ops for on-device training, whereas TensorFlow Micro does not. Additionally, the figure shows that TensorFlow Lite supports dynamic shapes and quantization-aware training, features that are absent in TensorFlow Micro. In contrast, both TensorFlow Lite and TensorFlow Micro offer native quantization tooling and support. Here, quantization refers to transforming an ML program into an approximated representation with available lower precision operations, a crucial feature for embedded and edge devices with limited computational resources.

6.9.2 Software

As shown in Figure 6.15, TensorFlow Lite Micro does not have OS support, while TensorFlow and TensorFlow Lite do. This design choice for TensorFlow Lite Micro helps reduce memory overhead, make startup times faster, and consume less energy. Instead, TensorFlow Lite Micro can be used in conjunction with real-time operating systems (RTOS) like FreeRTOS, Zephyr, and Mbed OS.

The figure also highlights an important memory management feature: TensorFlow Lite and TensorFlow Lite Micro support model memory mapping, allowing models to be directly accessed from flash storage rather than loaded into RAM. In contrast, TensorFlow does not offer this capability.



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

Figure 6.15: TensorFlow Framework Comparison - Software. Source: TensorFlow.

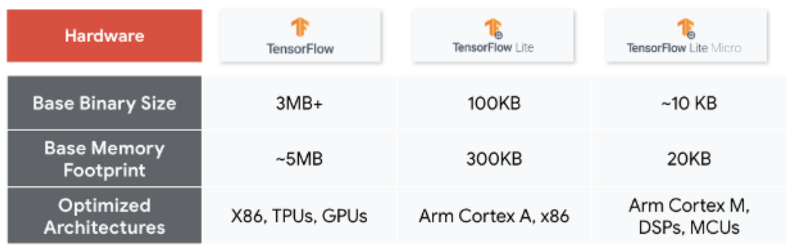
Another key difference is accelerator delegation. TensorFlow and TensorFlow Lite support this feature, allowing them to schedule code to different accelerators. However, TensorFlow Lite Micro does not offer accelerator delegation, as embedded systems tend to have a limited array of specialized accelerators.

These differences demonstrate how each TensorFlow variant is optimized for its target deployment environment, from powerful cloud servers to resource-constrained embedded devices.

6.9.3 Hardware

TensorFlow Lite and TensorFlow Lite Micro have significantly smaller base binary sizes and memory footprints than TensorFlow (see

Figure 6.16). For example, a typical TensorFlow Lite Micro binary is less than 200KB, whereas TensorFlow is much larger. This is due to the resource-constrained environments of embedded systems. TensorFlow supports x86, TPUs, and GPUs like NVIDIA, AMD, and Intel.



Hardware	TensorFlow	TensorFlow Lite	TensorFlow Lite Micro
Base Binary Size	3MB+	100KB	~10 KB
Base Memory Footprint	~5MB	300KB	20KB
Optimized Architectures	X86, TPUs, GPUs	Arm Cortex A, x86	Arm Cortex M, DSPs, MCUs

Figure 6.16: TensorFlow Framework Comparison - Hardware. Source: TensorFlow.

TensorFlow Lite supports Arm Cortex-A and x86 processors commonly used on mobile phones and tablets. The latter is stripped of all the unnecessary training logic for on-device deployment. TensorFlow Lite Micro provides support for microcontroller-focused Arm Cortex M cores like M0, M3, M4, and M7, as well as DSPs like Hexagon and SHARC and MCUs like STM32, NXP Kinetis, Microchip AVR.

6.9.4 Other Factors

Selecting the appropriate AI framework is essential to ensure that embedded systems can efficiently execute AI models. Several key factors beyond models, hardware, and software should be considered when evaluating AI frameworks for embedded systems.

Other key factors to consider when choosing a machine learning framework are performance, scalability, ease of use, integration with data engineering tools, integration with model optimization tools, and community support. Developers can make informed decisions and maximize the potential of your machine-learning initiatives by understanding these various factors.

6.9.4.1 Performance

Performance is critical in embedded systems where computational resources are limited. Evaluate the framework's ability to optimize model inference for embedded hardware. Model quantization and hardware acceleration support are crucial in achieving efficient inference.

6.9.4.2 Scalability

Scalability is essential when considering the potential growth of an embedded AI project. The framework should support the deployment of models on various embedded devices, from microcontrollers to more powerful processors. It should also seamlessly handle both small-scale and large-scale deployments.

6.9.4.3 Integration with Data Engineering Tools

Data engineering tools are essential for data preprocessing and pipeline management. An ideal AI framework for embedded systems should seamlessly integrate with these tools, allowing for efficient data ingestion, transformation, and model training.

6.9.4.4 Integration with Model Optimization Tools

Model optimization ensures that AI models are well-suited for embedded deployment. Evaluate whether the framework integrates with model optimization tools like TensorFlow Lite Converter or ONNX Runtime to facilitate model quantization and size reduction.

6.9.4.5 Ease of Use

The ease of use of an AI framework significantly impacts development efficiency. A framework with a user-friendly interface and clear documentation reduces developers' learning curve. Consideration should be given to whether the framework supports high-level APIs, allowing developers to focus on model design rather than low-level implementation details. This factor is incredibly important for embedded systems, which have fewer features than typical developers might be accustomed to.

6.9.4.6 Community Support

Community support plays another essential factor. Frameworks with active and engaged communities often have well-maintained codebases, receive regular updates, and provide valuable forums for problem-solving. As a result, community support also plays into Ease of Use because it ensures that developers have access to a wealth of resources, including tutorials and example projects. Community support provides some assurance that the framework will continue to be supported for future updates. There are only a few frameworks that cater to TinyML needs. TensorFlow Lite Micro is the most popular and has the most community support.

6.10 Future Trends in ML Frameworks

6.10.1 Decomposition

Currently, the ML system stack consists of four abstractions as shown in Figure 6.17, namely (1) computational graphs, (2) tensor programs, (3) libraries and runtimes, and (4) hardware primitives.

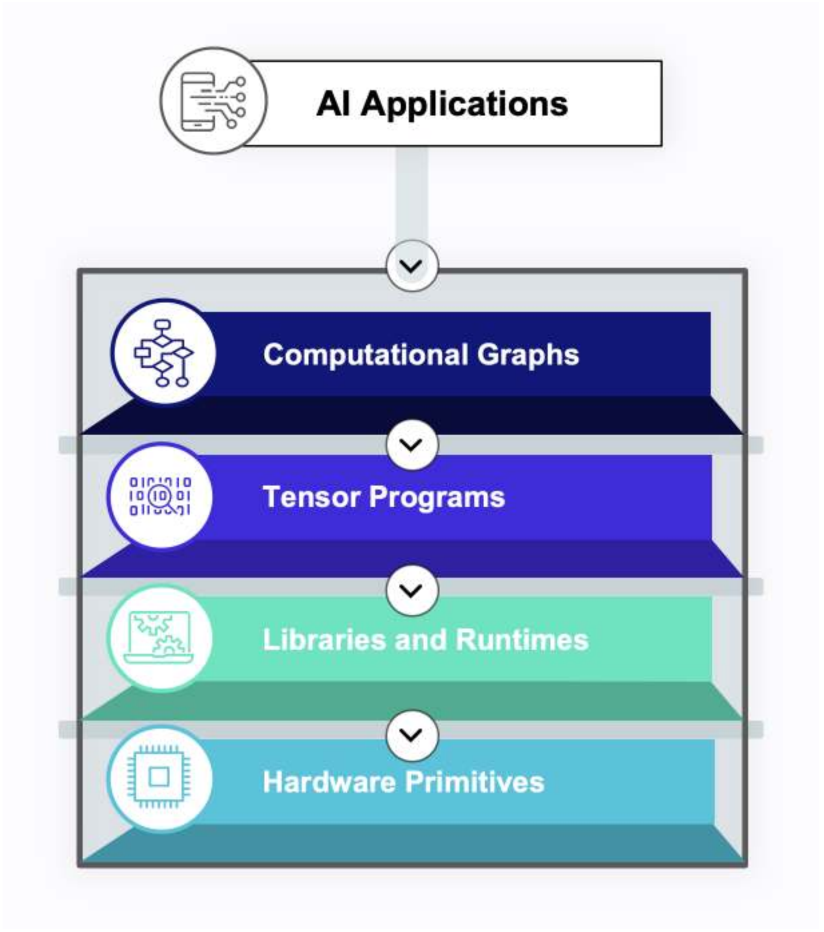


Figure 6.17: Four abstractions in current ML system stacks. Source: TVM.

This has led to vertical (i.e., between abstraction levels) and horizontal (i.e., library-driven vs. compilation-driven approaches to tensor computation) boundaries, which hinder innovation for ML. Future work in ML frameworks can look toward breaking these boundaries. In December 2021, Apache TVM Unity was proposed, which aimed to facilitate interactions between the different abstraction levels (as well as the people behind them, such as ML scientists, ML engineers, and hardware engineers) and co-optimize decisions in all four abstraction levels.

6.10.2 High-Performance Compilers & Libraries

As ML frameworks further develop, high-performance compilers and libraries will continue to emerge. Some current examples include TensorFlow XLA and Nvidia's CUTLASS, which accelerate linear algebra operations in computational graphs, and Nvidia's TensorRT, which accelerates and optimizes inference.

6.10.3 ML for ML Frameworks

We can also use ML to improve ML frameworks in the future. Some current uses of ML for ML frameworks include:

- Hyperparameter optimization using techniques such as Bayesian optimization, random search, and grid search
- Neural Architecture Search (NAS) to automatically search for optimal network architectures
- AutoML, which as described in Section 6.5, automates the ML pipeline.

6.11 Conclusion

In summary, selecting the optimal machine learning framework requires a thorough evaluation of various options against criteria such as usability, community support, performance, hardware compatibility, and model conversion capabilities. There is no one-size-fits-all solution, as the right framework depends on specific constraints and use cases.

We first introduced the necessity of machine learning frameworks like TensorFlow and PyTorch. These frameworks offer features such as tensors for handling multi-dimensional data, computational graphs for defining and optimizing model operations, and a suite of tools including loss functions, optimizers, and data loaders that streamline model development.

Advanced features further improve these frameworks' usability, enabling tasks like fine-tuning large pre-trained models and facilitating federated learning. These capabilities are critical for developing sophisticated machine learning models efficiently.

Embedded AI or TinyML frameworks, such as TensorFlow Lite Micro, provide specialized tools for deploying models on resource-constrained platforms. TensorFlow Lite Micro, for instance, offers comprehensive optimization tooling, including quantization mapping and kernel optimizations, to ensure high performance on microcontroller-based platforms like Arm Cortex-M and RISC-V processors. Frameworks specifically built for specialized hardware like CMSIS-NN on Cortex-M processors can further maximize performance but sacrifice portability. Integrated frameworks from processor vendors tailor the stack to their architectures, unlocking the full potential of their chips but locking you into their ecosystem.

Ultimately, choosing the right framework involves finding the best match between its capabilities and the requirements of the target platform. This requires balancing trade-offs between performance needs, hardware constraints, model complexity, and other factors. Thoroughly assessing the intended models and use cases and evaluating options against key metrics will guide developers in selecting the ideal framework for their machine learning applications.

6.12 Resources

Here is a curated list of resources to support students and instructors in their learning and teaching journeys. We are continuously working on expanding this collection and will add new exercises soon.

Slides

These slides are a valuable tool for instructors to deliver lectures and for students to review the material at their own pace. We encourage students and instructors to leverage these slides to improve their understanding and facilitate effective knowledge transfer.

- Frameworks overview.
- Embedded systems software.
- Inference engines: TF vs. TFLite.
- TF flavors: TF vs. TFLite vs. TFLite Micro.

- TFLite Micro:
  - TFLite Micro Big Picture.
  - TFLite Micro Interpreter.
  - TFLite Micro Model Format.
  - TFLite Micro Memory Allocation.
  - TFLite Micro NN Operations.

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.

- Exercise 9
- Exercise 10
- Exercise 11

Chapter 7

AI Training

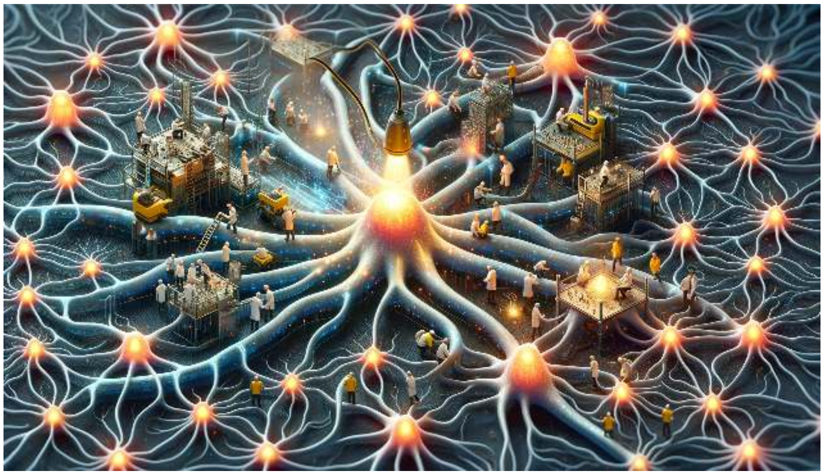


Figure 7.1: *DALL·E 3 Prompt: An illustration for AI training, depicting a neural network with neurons that are being repaired and firing. The scene includes a vast network of neurons, each glowing and firing to represent activity and learning. Among these neurons, small figures resembling engineers 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.*

Training is central to developing accurate and useful AI systems using machine learning techniques. At a high level, training involves feeding data into machine learning algorithms so they can learn patterns and make predictions. However, effectively training models requires tackling various challenges around data, algorithms, optimization of model parameters, and enabling generalization. This chapter will explore the nuances and considerations around training machine learning models.

Learning Objectives

- Understand the fundamental mathematics of neural networks, including linear transformations, activation functions, loss functions, backpropagation, and optimization via gradient descent.
- Learn how to effectively leverage data for model training through proper splitting into train, validation, and test sets to enable generalization.
- Learn various optimization algorithms like stochastic gradient descent and adaptations like momentum and Adam that accelerate training.
- Understand hyperparameter tuning and regularization techniques to improve model generalization by reducing overfitting.
- Learn proper weight initialization strategies matched to model architectures and activation choices that accelerate convergence.
- Identify the bottlenecks posed by key operations like matrix multiplication during training and deployment.
- Learn how hardware improvements like GPUs, TPUs, and specialized accelerators speed up critical math operations to accelerate training.
- Understand parallelization techniques, both data and model parallelism, to distribute training across multiple devices and accelerate system throughput.

7.1 Overview

Training is critical for developing accurate and useful AI systems using machine learning. The training creates a machine learning model that can generalize to new, unseen data rather than memorizing the training examples. This is done by feeding training data into algorithms that learn patterns from these examples by adjusting internal parameters.

The algorithms minimize a loss function, which compares their predictions on the training data to the known labels or solutions, guiding the learning. Effective training often requires high-quality, representative data sets large enough to capture variability in real-world use cases.

It also requires choosing an algorithm suited to the task, whether a neural network for computer vision, a reinforcement learning algorithm for robotic control, or a tree-based method for categorical prediction. Careful tuning is needed for the model structure, such as neural network depth and width, and learning parameters like step size and regularization strength.

Techniques to prevent overfitting like regularization penalties and validation with held-out data, are also important. Overfitting can occur when a model fits the training data too closely, failing to generalize to new data. This can happen if the model is too complex or trained too long.

To avoid overfitting, regularization techniques can help constrain the model. One regularization method is adding a penalty term to the loss function that discourages complexity, like the L2 norm of the weights. This penalizes large parameter values. Another technique is dropout, where a percentage of neurons is randomly set to zero during training. This reduces neuron co-adaptation.

Validation methods also help detect and avoid overfitting. Part of the training data is held out from the training loop as a validation set. The model is evaluated on this data. If validation error increases while training error decreases, overfitting occurs. The training can then be stopped early or regularized more strongly. Regularization and validation enable models to train to maximum capability without overfitting the training data.

Training takes significant computing resources, especially for deep neural networks used in computer vision, natural language processing, and other areas. These networks have millions of adjustable weights that must be tuned through extensive training. Hardware improvements and distributed training techniques have enabled training ever larger neural nets that can achieve human-level performance on some tasks.

In summary, some key points about training:

- **Data is crucial:** Machine learning models learn from examples in training data. More high-quality, representative data leads to better model performance. Data needs to be processed and formatted for training.
- **Algorithms learn from data:** Different algorithms (neural networks, decision trees, etc.) have different approaches to finding patterns in data. Choosing the right algorithm for the task is important.

- **Training refines model parameters:** Model training adjusts internal parameters to find patterns in data. Advanced models like neural networks have many adjustable weights. Training iteratively adjusts weights to minimize a loss function.
- **Generalization is the goal:** A model that overfits the training data will not generalize well. Regularization techniques (dropout, early stopping, etc.) reduce overfitting. Validation data is used to evaluate generalization.
- **Training takes compute resources:** Training complex models requires significant processing power and time. Hardware improvements and distributed training across GPUs/TPUs have enabled advances.

We will walk you through these details in the rest of the sections. Understanding how to effectively leverage data, algorithms, parameter optimization, and generalization through thorough training is essential for developing capable, deployable AI systems that work robustly in the real world.

7.2 Mathematics of Neural Networks

Deep learning has revolutionized machine learning and artificial intelligence, enabling computers to learn complex patterns and make intelligent decisions. The neural network is at the heart of the deep learning revolution, and as discussed in section 3, "Deep Learning Primer," it is a cornerstone in some of these advancements.

Neural networks are made up of simple functions layered on each other. Each layer takes in some data, performs some computation, and passes it to the next layer. These layers learn progressively high-level features useful for the tasks the network is trained to perform. For example, in a network trained for image recognition, the input layer may take in pixel values, while the next layers may detect simple shapes like edges. The layers after that may detect more complex shapes like noses, eyes, etc. The final output layer classifies the image as a whole.

The network in a neural network refers to how these layers are connected. Each layer's output is considered a set of neurons, which are connected to neurons in the subsequent layers, forming a "network." The way these neurons interact is determined by the weights between them, which model synaptic strengths similar to that of a brain's neuron. The neural network is trained by adjusting these weights. Concretely, the weights are initially set randomly, then input is fed in, the output is compared to the desired result, and finally, the weights are tweaked to improve the network. This process is repeated until the network reliably minimizes the loss, indicating it has learned the patterns in the data.

How is this process defined mathematically? Formally, neural networks are mathematical models that consist of alternating linear and nonlinear operations, parameterized by a set of learnable weights that are trained to minimize some loss function. This loss function measures how good our model is concerning fitting our training data, and it produces a numerical value when evaluated on our model against the training data. Training neural networks involves repeatedly evaluating the loss function on many different data points to measure how good our model is, then continuously tweaking the weights of our model using backpropagation so that the loss decreases, ultimately optimizing the model to fit our data.

7.2.1 Neural Network Notation

The core of a neural network can be viewed as a sequence of alternating linear and nonlinear operations, as shown in Figure 7.2.

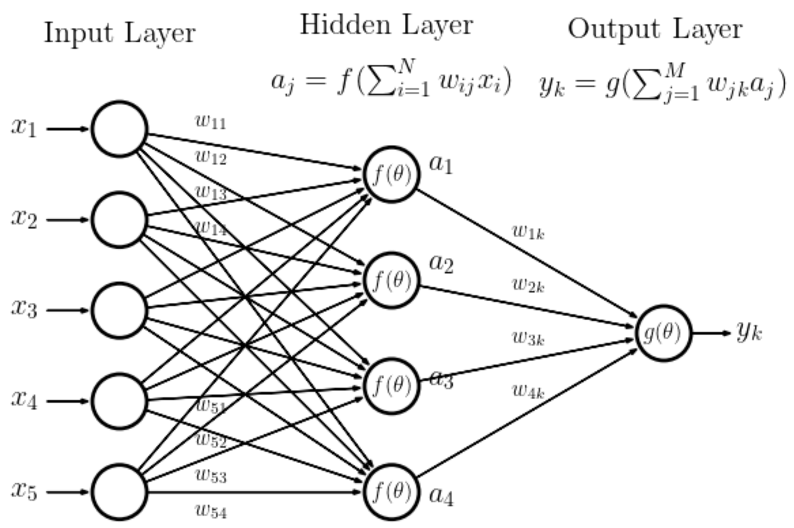


Figure 7.2: Neural network diagram. Source: astroML.

Neural networks are structured with layers of neurons connected by weights (representing linear operations) and activation functions (representing nonlinear operations). By examining the figure, we see how information flows through the network, starting from the input layer, passing through one or more hidden layers, and finally reaching the output layer. Each connection between neurons represents a weight, while each neuron typically applies a nonlinear activation function to its inputs.

The neural network operates by taking an input vector x_i and passing it through a series of layers, each of which performs linear and non-linear operations. The output of the network at each layer A_j can be represented as:

$$A_j = f\left(\sum_{i=1}^{N} w_{ij} x_i\right)$$

Where:

- N - The total number of input features.
- x_i - The individual input feature, where i ranges from 1 to N.
- w_{ij} - The weights connecting neuron i in one layer to neuron j in the next layer, which are adjusted during training.
- $f(\theta)$ - The non-linear activation function applied at each layer (e.g., ReLU, softmax, etc.).
- A_j - The output of the neural network at each layer j, where j denotes the layer number.

In the context of Figure 7.2, x_1, x_2, x_3, x_4, and x_5 represent the input features. Each input neuron x_i corresponds to one feature of the input data. The arrows from the input layer to the hidden layer indicate connections between the input neurons and the hidden neurons, with each connection associated with a weight w_{ij}.

The hidden layer consists of neurons a_1, a_2, a_3, and a_4, each receiving input from all the neurons in the input layer. The weights w_{ij} connect the input neurons to the hidden neurons. For example, w_{11} is the weight connecting input x_1 to hidden neuron a_1.

The number of nodes in each layer and the total number of layers together define the architecture of the neural network. In the first layer (input layer), the number of nodes corresponds to the dimensionality of the input data, while in the last layer (output layer), the number of nodes corresponds to the dimensionality of the output. The number of nodes in the intermediate layers can be set arbitrarily, allowing flexibility in designing the network architecture.

The weights, which determine how each layer of the neural network interacts with the others, are matrices of real numbers. Additionally, each layer typically includes a bias vector, but we are ignoring it here for simplicity. The weight matrix W_j connecting layer $j-1$ to layer j has the dimensions:

$$W_j \in \mathbb{R}^{d_j \times d_{j-1}}$$

where d_j is the number of nodes in layer j, and d_{j-1} is the number of nodes in the previous layer $j-1$.

The final output y_k of the network is obtained by applying another activation function $g(\theta)$ to the weighted sum of the hidden layer outputs:

$$y = g\left(\sum_{j=1}^{M} w_{jk} A_j\right)$$

Where:

- M - The number of hidden neurons in the final layer before the output.
- w_{jk} - The weight between hidden neuron a_j and output neuron y_k.
- $g(\theta)$ - The activation function applied to the weighted sum of the hidden layer outputs.

Our neural network, as defined, performs a sequence of linear and nonlinear operations on the input data (x_i) to obtain predictions (y_i), which hopefully is a good answer to what we want the neural network to do on the input (i.e., classify if the input image is a cat or not). Our neural network may then be represented succinctly as a function N which takes in an input $x \in \mathbb{R}^{d_0}$ parameterized by $W_1, ..., W_n$, and produces the final output y:

$$y = N(x; W_1, ..., W_n) \quad \text{where } A_0 = x$$

This equation indicates that the network starts with the input $A_0 = x$ and iteratively computes A_j at each layer using the parameters W_j until it produces the final output y at the output layer.

Next, we will see how to evaluate this neural network against training data by introducing a loss function.

Note

Why are the nonlinear operations necessary? If we only had linear layers, the entire network would be equivalent to a single linear layer consisting of the product of the linear operators. Hence, the nonlinear functions play a key role in the power of neural networks as they improve the neural network's ability to fit functions.

ℹ Note

Convolutions are also linear operators and can be cast as a matrix multiplication.

7.2.2 Loss Function as a Measure of Goodness of Fit against Training Data

After defining our neural network, we are given some training data, which is a set of points (x_j, y_j) for $j = 1 \to M$, where M is the total number of samples in the dataset, and j indexes each sample. We want to evaluate how good our neural network is at fitting this data. To do this, we introduce a loss function, which is a function that takes the output of the neural network on a particular datapoint $\hat{y}_j = N(x_j; W_1, \ldots, W_n)$ and compares it against the "label" of that particular datapoint (the corresponding y_j), and outputs a single numerical scalar (i.e., one real number) that represents how "good" the neural network fits that particular data point; the final measure of how good the neural network is on the entire dataset is therefore just the average of the losses across all data points.

There are many different types of loss functions; for example, in the case of image classification, we might use the cross-entropy loss function, which tells us how well two vectors representing classification predictions compare (i.e., if our prediction predicts that an image is more likely a dog, but the label says it is a cat, it will return a high "loss," indicating a bad fit).

Mathematically, a loss function is a function that takes in two real-valued vectors, one representing the predicted outputs of the neural network and the other representing the true labels, and outputs a single numerical scalar representing the error or "loss."

$$L : \mathbb{R}^{d_n} \times \mathbb{R}^{d_n} \longrightarrow \mathbb{R}$$

For a single training example, the loss is given by:

$$L(N(x_j; W_1, \ldots, W_n), y_j)$$

where $\hat{y}_j = N(x_j; W_1, \ldots, W_n)$ is the predicted output of the neural network for the input x_j, and y_j is the true label.

The total loss across the entire dataset, L_{full}, is then computed as the average loss across all data points in the training data:

Loss Function for Optimizing Neural Network Model on a Dataset

$$L_{full} = \frac{1}{M}\sum_{j=1}^{M} L(N(x_j; W_1, ...W_n), y_j)$$

7.2.3 Training Neural Networks with Gradient Descent

Now that we can measure how well our network fits the training data, we can optimize the neural network weights to minimize this loss. In this context, we are denoting W_i as the weights for each layer i in the network. At a high level, we tweak the parameters of the real-valued matrices W_is to minimize the loss function L_{full}. Overall, our mathematical objective is

Neural Network Training Objective

$$min_{W_1,...,W_n} L_{full}$$

$$= min_{W_1,...,W_n} \frac{1}{M}\sum_{j=1}^{M} L(N(x_j; W_1, ...W_n), y_j)$$

So, how do we optimize this objective? Recall from calculus that minimizing a function can be done by taking the function's derivative concerning the input parameters and tweaking the parameters in the gradient direction. This technique is called gradient descent and concretely involves calculating the derivative of the loss function L_{full} concerning $W_1, ..., W_n$ to obtain a gradient for these parameters to take a step in, then updating these parameters in the direction of the gradient. Thus, we can train our neural network using gradient descent, which repeatedly applies the update rule.

Gradient Descent Update Rule

$$W_i := W_i - \lambda \frac{\partial L_{full}}{\partial W_i} \text{ for } i = 1..n$$

Note

In practice, the gradient is computed over a minibatch of data points to improve computational efficiency. This is called stochastic gradient descent or batch gradient descent.

Where λ is the stepsize or learning rate of our tweaks, in training our neural network, we repeatedly perform the step above until convergence, or when the loss no longer decreases. Figure 7.3 illustrates this process: we want to reach the minimum point, which's done by following the gradient (as illustrated with the blue arrows in the figure). This prior approach is known as full gradient descent since we are computing the derivative concerning the entire training data and only then taking a single gradient step; a more efficient approach is to calculate the gradient concerning just a random batch of data points and then taking a step, a process known as batch gradient descent or stochastic gradient descent (Robbins and Monro 1951), which is more efficient since now we are taking many more steps per pass of the entire training data. Next, we will cover the mathematics behind computing the gradient of the loss function concerning the W_is, a process known as backpropagation.

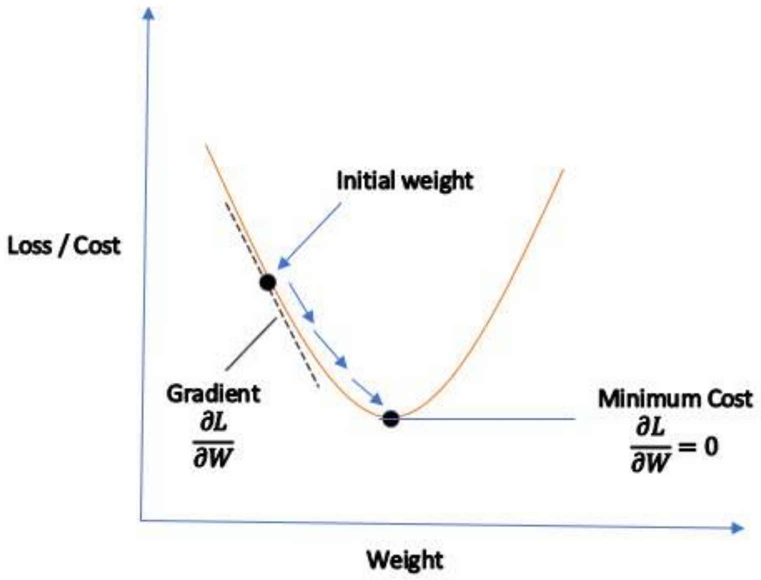

Loss / Cost
Initial weight
Gradient
$\frac{\partial L}{\partial W}$
Minimum Cost
$\frac{\partial L}{\partial W} = 0$
Weight

Figure 7.3: Gradient descent. Source: Towards Data Science.

7.2.4 Backpropagation

Training neural networks involve repeated applications of the gradient descent algorithm, which involves computing the derivative of the loss function with respect to the W_is. How do we compute the loss derivative concerning the W_is, given that the W_is are nested functions of each other in a deep neural network? The trick is to leverage the **chain rule:** we can compute the derivative of the loss concerning the W_is by repeatedly applying the chain rule in a complete process known as backpropagation. Specifically, we can calculate the gradients by computing the derivative of the loss concerning the outputs of the last layer, then progressively use this to compute the derivative of the loss concerning each prior layer to the input layer. This process starts from the end of the network (the layer closest to the output) and progresses backwards, and hence gets its name backpropagation.

Let's break this down. We can compute the derivative of the loss concerning the *outputs of each layer of the neural network* by using repeated applications of the chain rule.

$$\frac{\partial L_{full}}{\partial L_n} = \frac{\partial A_n}{\partial L_n}\frac{\partial L_{full}}{\partial A_n}$$

$$\frac{\partial L_{full}}{\partial L_{n-1}} = \frac{\partial A_{n-1}}{\partial L_{n-1}}\frac{\partial L_n}{\partial A_{n-1}}\frac{\partial A_n}{\partial L_n}\frac{\partial L_{full}}{\partial A_n}$$

or more generally

$$\frac{\partial L_{full}}{\partial L_i} = \frac{\partial A_i}{\partial L_i}\frac{\partial L_{i+1}}{\partial A_i}\dots\frac{\partial A_n}{\partial L_n}\frac{\partial L_{full}}{\partial A_n}$$

Note

In what order should we perform this computation? From a computational perspective, performing the calculations from the end to the front is preferable. (i.e: first compute $\frac{\partial L_{full}}{\partial A_n}$ then the prior terms, rather than start in the middle) since this avoids materializing and computing large jacobians. This is because $\frac{\partial L_{full}}{\partial A_n}$ is a vector; hence, any matrix operation that includes this term has an output that is squished to be a vector. Thus, performing the computation from the end avoids large matrix-matrix multiplications by ensuring that the intermediate products are vectors.

Note

In our notation, we assume the intermediate activations A_i are *column* vectors, rather than *row* vectors, hence the chain rule is $\frac{\partial L}{\partial L_i} = \frac{\partial L_{i+1}}{\partial L_i}\dots\frac{\partial L}{\partial L_n}$ rather than $\frac{\partial L}{\partial L_i} = \frac{\partial L}{\partial L_n}\dots\frac{\partial L_{i+1}}{\partial L_i}$

After computing the derivative of the loss concerning the *output of each layer*, we can easily obtain the derivative of the loss concerning the *parameters*, again using the chain rule:

$$\frac{\partial L_{full}}{W_i} = \frac{\partial L_i}{\partial W_i}\frac{\partial L_{full}}{\partial L_i}$$

And this is ultimately how the derivatives of the layers' weights are computed using backpropagation! What does this concretely look like in a specific example? Below, we walk through a specific example of a simple 2-layer neural network on a regression task using an MSE loss function with 100-dimensional inputs and a 30-dimensional hidden layer:

Example of Backpropagation
Suppose we have a two-layer neural network

$$L_1 = W_1 A_0$$

$$A_1 = ReLU(L_1)$$

$$L_2 = W_2 A_1$$

$$A_2 = ReLU(L_2)$$

$$NN(x) = \text{Let } A_0 = x \text{ then output } A_2$$

where $W_1 \in \mathbb{R}^{30\times 100}$ and $W_2 \in \mathbb{R}^{1\times 30}$. Furthermore, suppose we use the MSE loss function:

$$L(x,y) = (x-y)^2$$

We wish to compute

$$\frac{\partial L(NN(x),y)}{\partial W_i} \text{ for } i = 1,2$$

Note the following:

$$\frac{\partial L(x,y)}{\partial x} = 2\times(x-y)$$

$$\frac{\partial ReLU(x)}{\partial x}\delta = \left\{\begin{array}{ll} 0 & \text{for } x \le 0 \\ 1 & \text{for } x \ge 0 \end{array}\right\} \odot \delta$$

$$\frac{\partial WA}{\partial A}\delta = W^T\delta$$

$$\frac{\partial WA}{\partial W}\delta = \delta A^T$$

Then we have

$$\frac{\partial L(NN(x),y)}{\partial W_2} = \frac{\partial L_2}{\partial W_2}\frac{\partial A_2}{\partial L_2}\frac{\partial L(NN(x),y)}{\partial A_2}$$
$$= (2L(NN(x)-y) \odot ReLU'(L_2))A_1^T$$

and

$$\frac{\partial L(NN(x),y)}{\partial W_1} = \frac{\partial L_1}{\partial W_1}\frac{\partial A_1}{\partial L_1}\frac{\partial L_2}{\partial A_1}\frac{\partial A_2}{\partial L_2}\frac{\partial L(NN(x),y)}{\partial A_2}$$
$$= [ReLU'(L_1) \odot (W_2^T[2L(NN(x)-y) \odot ReLU'(L_2)])]A_0^T$$

Tip

Double-check your work by making sure that the shapes are correct!

- All Hadamard products ($\odot$) should operate on tensors of the same shape
- All matrix multiplications should operate on matrices that share a common dimension (i.e., m by n, n by k)
- All gradients concerning the weights should have the same shape as the weight matrices themselves

The entire backpropagation process can be complex, especially for very deep networks. Fortunately, machine learning frameworks like PyTorch support automatic differentiation, which performs backpropagation for us. In these frameworks, we simply need to specify the forward pass, and the derivatives will be automatically computed for us. Nevertheless, it is beneficial to understand the theoretical process that is happening under the hood in these machine-learning frameworks.

Note

As seen above, intermediate activations A_i are reused in backpropagation. To improve performance, these activations are cached from the forward pass to avoid being recomputed. However, activations must be kept in memory between the forward and backward passes, leading to higher memory usage. If the network and batch size are large, this may lead to memory issues. Similarly, the derivatives with respect to each layer's outputs are cached to avoid recomputation.

Caution 12: Neural Networks with Backpropagation and Gradient Descent

Unlock the math behind powerful neural networks! Deep learning might seem like magic, but it's rooted in mathematical principles. In this chapter, you've broken down neural network notation, loss functions, and the powerful technique of backpropagation. Now, prepare to implement this theory with these Colab notebooks. Dive into the heart of how neural networks learn. You'll see the math behind backpropagation and gradient descent, updating those weights step-by-step.

Open in Colab

7.3 Differentiable Computation Graphs

In general, stochastic gradient descent using backpropagation can be performed on any computational graph that a user may define, provided that the operations of the computation are differentiable. As such, generic deep learning libraries like PyTorch and Tensorflow allow users to specify their computational process (i.e., neural networks) as a computational graph. Backpropagation is automatically performed via automatic differentiation when stochastic gradient descent is performed on these computational graphs. Framing AI training as an optimization problem on differentiable computation graphs is a general way to understand what is happening under the hood with deep learning systems.

The structure depicted in Figure 7.4 showcases a segment of a differentiable computational graph. In this graph, the input 'x' is processed through a series of operations: it is first multiplied by a weight matrix 'W' (MatMul), then added to a bias 'b' (Add), and finally passed to an activation function, Rectified Linear Unit (ReLU). This sequence of operations gives us the output C. The graph's differentiable nature means that each operation has a well-defined gradient. Automatic differentiation, as implemented in ML frameworks, leverages this property to efficiently compute the gradients of the loss with respect to each parameter in the network (e.g., 'W' and 'b').

7.4 Training Data

To enable effective neural network training, the available data must be split into training, validation, and test sets. The training set is used

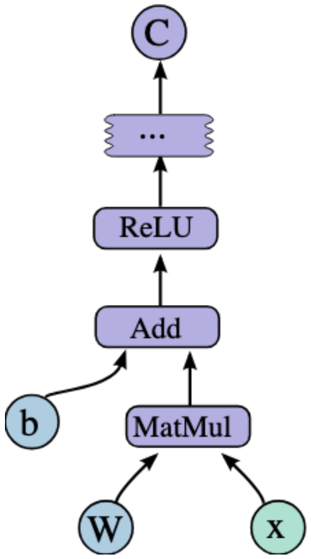


Figure 7.4: Computational Graph. Source: TensorFlow.

to train the model parameters. The validation set evaluates the model during training to tune hyperparameters and prevent overfitting. The test set provides an unbiased final evaluation of the trained model's performance.

Maintaining clear splits between train, validation, and test sets with representative data is crucial to properly training, tuning, and evaluating models to achieve the best real-world performance. To this end, we will learn about the common pitfalls or mistakes people make when creating these data splits.

Table 7.1 compares the differences between training, validation, and test data splits:

Table 7.1: Comparing training, validation, and test data splits.

Data Split	Purpose	Typical Size
Training Set	Train the model parameters	60-80% of total data
Validation Set	Evaluate model during training to tune hyperparameters and prevent overfitting	20% of total data
Test Set	Provide unbiased evaluation of final trained model	20% of total data

7.4.1 Dataset Splits

7.4.1.1 Training Set

The training set is used to train the model. It is the largest subset, typically 60-80% of the total data. The model sees and learns from the training data to make predictions. A sufficiently large and representative training set is required for the model to learn the underlying patterns effectively.

7.4.1.2 Validation Set

The validation set evaluates the model during training, usually after each epoch. Typically, 20% of the data is allocated for the validation set. The model does not learn or update its parameters based on the validation data. It is used to tune hyperparameters and make other tweaks to improve training. Monitoring metrics like loss and accuracy on the validation set prevents overfitting on just the training data.

7.4.1.3 Test Set

The test set acts as a completely unseen dataset that the model did not see during training. It is used to provide an unbiased evaluation of the final trained model. Typically, 20% of the data is reserved for testing. Maintaining a hold-out test set is vital for obtaining an accurate estimate of how the trained model would perform on real-world unseen data. Data leakage from the test set must be avoided at all costs.

The relative proportions of the training, validation, and test sets can vary based on data size and application. However, following the general guidelines for a 60/20/20 split is a good starting point. Careful data splitting ensures models are properly trained, tuned, and evaluated to achieve the best performance.

Video 5 explains how to properly split the dataset into training, validation, and testing sets, ensuring an optimal training process.

! Important 5: Train/Dev/Test Sets

https://www.youtube.com/watch?v=1waHlpKiNyY

7.4.2 Common Pitfalls and Mistakes

7.4.2.1 Insufficient Training Data

Allocating too little data to the training set is a common mistake when splitting data that can severely impact model performance. If the training set is too small, the model will not have enough samples to effectively learn the true underlying patterns in the data. This leads to high variance and causes the model to fail to generalize well to new data.

For example, if you train an image classification model to recognize handwritten digits, providing only 10 or 20 images per digit class would be completely inadequate. The model would need more examples to capture the wide variances in writing styles, rotations, stroke widths, and other variations.

As a rule of thumb, the training set size should be at least hundreds or thousands of examples for most machine learning algorithms to work effectively. Due to the large number of parameters, the training set often needs to be in the tens or hundreds of thousands for deep neural networks, especially those using convolutional layers.

Insufficient training data typically manifests in symptoms like high error rates on validation/test sets, low model accuracy, high variance, and overfitting on small training set samples. Collecting more quality training data is the solution. Data augmentation techniques can also help virtually increase the size of training data for images, audio, etc.

Carefully factoring in the model complexity and problem difficulty when allocating training samples is important to ensure sufficient data is available for the model to learn successfully. Following guidelines on minimum training set sizes for different algorithms is also recommended. More training data is needed to maintain the overall success of any machine learning application.

Consider Figure 7.5 where we try to classify/split datapoints into two categories (here, by color): On the left, overfitting is depicted by a model that has learned the nuances in the training data too well (either the dataset was too small or we ran the model for too long), causing it to follow the noise along with the signal, as indicated by the line's excessive curves. The right side shows underfitting, where the model's simplicity prevents it from capturing the dataset's underlying structure, resulting in a line that does not fit the data well. The center graph represents an ideal fit, where the model balances well between generalization and fitting, capturing the main trend of the data without being swayed by outliers. Although the model is not a perfect fit (it misses some points), we care more about its ability to recognize general patterns rather than idiosyncratic outliers.

Figure 7.6 illustrates the process of fitting the data over time. When training, we search for the "sweet spot" between underfitting and overfitting. At first when the model hasn't had enough time to learn the pat-

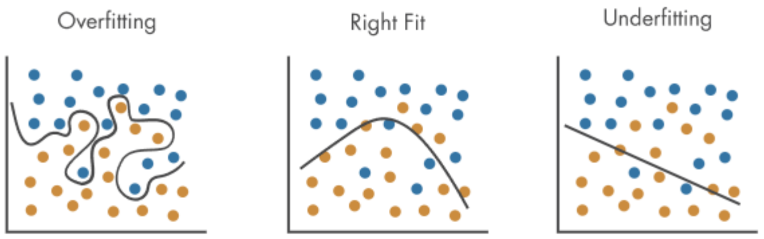


Figure 7.5: Data fitting: overfitting, right fit, and underfitting. Source: MathWorks.

terns in the data, we find ourselves in the underfitting zone, indicated by high error rates on the validation set (remember that the model is trained on the training set and we test its generalizability on the validation set, or data it hasn't seen before). At some point, we achieve a global minimum for error rates, and ideally we want to stop the training there. If we continue training, the model will start "memorizing" or getting to know the data too well that the error rate starts going back up, since the model will fail to generalize to data it hasn't seen before.

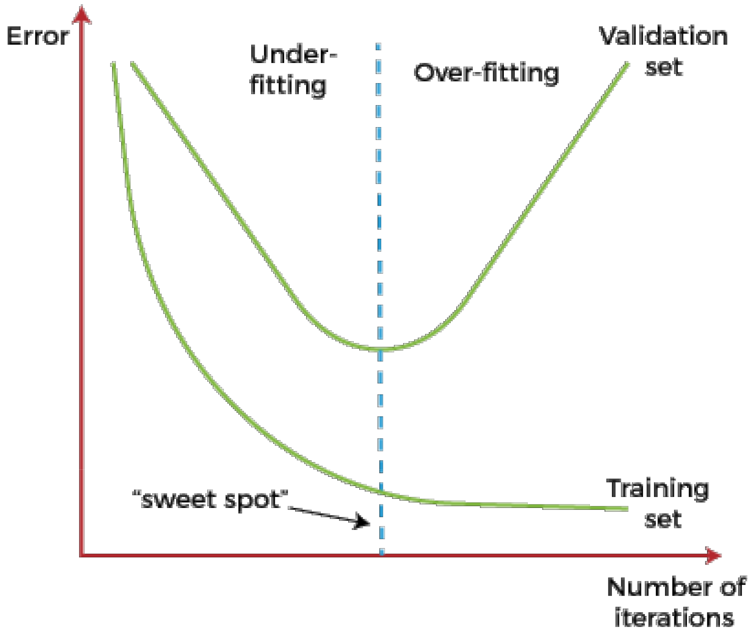


Figure 7.6: Fitting the data overtime. Source: IBM.

Video 6 provides an overview of bias and variance and the relationship between the two concepts and model accuracy.

! Important 6: Bias/Variance

https://www.youtube.com/watch?v=SjQyLhQIXSM

7.4.2.2 Data Leakage Between Sets

Data leakage refers to the unintentional transfer of information between the training, validation, and test sets. This violates the fundamental assumption that the splits are mutually exclusive. Data leakage leads to seriously compromised evaluation results and inflated performance metrics.

A common way data leakage occurs is if some samples from the test set are inadvertently included in the training data. When evaluating the test set, the model has already seen some of the data, which gives overly optimistic scores. For example, if 2% of the test data leaks into the training set of a binary classifier, it can result in an accuracy boost of up to 20%!

If the data splits are not done carefully, more subtle forms of leakage can happen. If the splits are not properly randomized and shuffled, samples that are close to each other in the dataset may end up in the same split, leading to distribution biases. This creates information bleed through based on proximity in the dataset.

Another case is when datasets have linked, inherently connected samples, such as graphs, networks, or time series data. Naive splitting may isolate connected nodes or time steps into different sets. Models can make invalid assumptions based on partial information.

Preventing data leakage requires creating solid separation between splits—no sample should exist in more than one split. Shuffling and randomized splitting help create robust divisions. Cross-validation techniques can be used for more rigorous evaluation. Detecting leakage is difficult, but telltale signs include models doing way better on test vs. validation data.

Data leakage severely compromises the validity of the evaluation because the model has already partially seen the test data. No amount of tuning or complex architectures can substitute for clean data splits. It is better to be conservative and create complete separation between splits to avoid this fundamental mistake in machine learning pipelines.

7.4.2.3 Small or Unrepresentative Validation Set

The validation set is used to assess model performance during training and to fine-tune hyperparameters. For reliable and stable evaluations, the validation set should be sufficiently large and representative of the real data distribution. However, this can make model selection and tuning more challenging.

For example, if the validation set only contains 100 samples, the metrics calculated will have a high variance. Due to noise, the accuracy may fluctuate up to 5-10% between epochs. This makes it difficult to know if a drop in validation accuracy is due to overfitting or natural variance. With a larger validation set, say 1000 samples, the metrics will be much more stable.

Additionally, if the validation set is not representative, perhaps missing certain subclasses, the estimated skill of the model may be inflated. This could lead to poor hyperparameter choices or premature training stops. Models selected based on such biased validation sets do not generalize well to real data.

A good rule of thumb is that the validation set size should be at least several hundred samples and up to 10-20% of the training set, while still leaving sufficient samples for training. The splits should also be stratified, meaning that the class proportions in the validation set should match those in the full dataset, especially if working with imbalanced datasets. A larger validation set representing the original data characteristics is essential for proper model selection and tuning.

7.4.2.4 Reusing the Test Set Multiple Times

The test set is designed to provide an unbiased evaluation of the fully trained model only once at the end of the model development process. Reusing the test set multiple times during development for model evaluation, hyperparameter tuning, model selection, etc., can result in overfitting on the test data. Instead, reserve the test set for a final evaluation of the fully trained model, treating it as a black box to simulate its performance on real-world data. This approach provides reliable metrics to determine whether the model is ready for production deployment.

If the test set is reused as part of the validation process, the model may start to see and learn from the test samples. This, coupled with intentionally or unintentionally optimizing model performance on the test set, can artificially inflate metrics like accuracy.

For example, suppose the test set is used repeatedly for model selection out of 5 architectures. In that case, the model may achieve 99% test accuracy by memorizing the samples rather than learning generalizable patterns. However, when deployed in the real world, the accuracy of new data could drop by 60%.

The best practice is to interact with the test set only once at the end to report unbiased metrics on how the final tuned model would perform in the real world. While developing the model, the validation set should be used for all parameter tuning, model selection, early stopping, and similar tasks. It's important to reserve a portion, such as 20-30% of the full dataset, solely for the final model evaluation. This data should not be used for validation, tuning, or model selection during development.

Failing to keep an unseen hold-out set for final validation risks optimizing results and overlooking potential failures before model release. Having some fresh data provides a final sanity check on real-world efficacy. Maintaining the complete separation of training/validation from the test set is essential to obtain accurate estimates of model performance. Even minor deviations from a single use of the test set could positively bias results and metrics, providing an overly optimistic view of real-world efficacy.

7.4.2.5 Same Data Splits Across Experiments

When comparing different machine learning models or experimenting with various architectures and hyperparameters, using the same data splits for training, validation, and testing across the different experiments can introduce bias and invalidate the comparisons.

If the same splits are reused, the evaluation results may be more balanced and accurately measure which model performs better. For example, a certain random data split may favor model A over model B irrespective of the algorithms. Reusing this split will then bias towards model A.

Instead, the data splits should be randomized or shuffled for each experimental iteration. This ensures that randomness in the sampling of the splits does not confer an unfair advantage to any model.

With different splits per experiment, the evaluation becomes more robust. Each model is tested on a wide range of test sets drawn randomly from the overall population, smoothing out variation and removing correlation between results.

Proper practice is to set a random seed before splitting the data for each experiment. Splitting should occur after shuffling/resampling as part of the experimental pipeline. Carrying out comparisons on the same splits violates the i.i.d (independent and identically distributed) assumption required for statistical validity.

Unique splits are essential for fair model comparisons. Though more compute-intensive, randomized allocation per experiment removes sampling bias and enables valid benchmarking. This highlights the true differences in model performance irrespective of a particular split's characteristics.

7.4.2.6 Failing to Stratify Splits

When splitting data into training, validation, and test sets, failing to stratify the splits can result in an uneven representation of the target classes across the splits and introduce sampling bias. This is especially problematic for imbalanced datasets.

Stratified splitting involves sampling data points such that the proportion of output classes is approximately preserved in each split. For example, if performing a 70/30 train-test split on a dataset with 60% negative and 40% positive samples, stratification ensures ~60% negative and ~40% positive examples in both training and test sets.

Without stratification, random chance could result in the training split having 70% positive samples while the test has 30% positive samples. The model trained on this skewed training distribution will not generalize well. Class imbalance also compromises model metrics like accuracy.

Stratification works best when done using labels, though proxies like clustering can be used for unsupervised learning. It becomes essential for highly skewed datasets with rare classes that could easily be omitted from splits.

Libraries like Scikit-Learn have stratified splitting methods built into them. Failing to use them could inadvertently introduce sampling bias and hurt model performance on minority groups. After performing the splits, the overall class balance should be examined to ensure even representation across the splits.

Stratification provides a balanced dataset for both model training and evaluation. Though simple random splitting is easy, mindful of stratification needs, especially for real-world imbalanced data, results in more robust model development and evaluation.

7.4.2.7 Ignoring Time Series Dependencies

Time series data has an inherent temporal structure with observations depending on past context. Naively splitting time series data into train and test sets without accounting for this dependency leads to data leakage and lookahead bias.

For example, simply splitting a time series into the first 70% of training and the last 30% as test data will contaminate the training data with future data points. The model can use this information to "peek" ahead during training.

This results in an overly optimistic evaluation of the model's performance. The model may appear to forecast the future accurately but has actually implicitly learned based on future data, which does not translate to real-world performance.

Proper time series cross-validation techniques, such as forward chaining, should be used to preserve order and dependency. The test set should only contain data points from a future time window that the model was not exposed to for training.

Failing to account for temporal relationships leads to invalid causality assumptions. If the training data contains future points, the model may also need to learn how to extrapolate forecasts further.

Maintaining the temporal flow of events and avoiding lookahead bias is key to properly training and testing time series models. This ensures they can truly predict future patterns and not just memorize past training data.

7.4.2.8 No Unseen Data for Final Evaluation

A common mistake when splitting data is failing to set aside some portion of the data just for the final evaluation of the completed model. All of the data is used for training, validation, and test sets during development.

This leaves no unseen data to get an unbiased estimate of how the final tuned model would perform in the real world. The metrics on the test set used during development may only partially reflect actual model skills.

For example, choices like early stopping and hyperparameter tuning are often optimized based on test set performance. This couples the model to the test data. An unseen dataset is needed to break this coupling and get true real-world metrics.

Best practice is to reserve a portion, such as 20-30% of the full dataset, solely for final model evaluation. This data should not be used for validation, tuning, or model selection during development.

Saving some unseen data allows for evaluating the completely trained model as a black box on real-world data. This provides reliable metrics to decide whether the model is ready for production deployment.

Failing to keep an unseen hold-out set for final validation risks optimizing results and overlooking potential failures before model release. Having some fresh data provides a final sanity check on real-world efficacy.

7.4.2.9 Overoptimizing on the Validation Set

The validation set is meant to guide the model training process, not serve as additional training data. Overoptimizing the validation set to maximize performance metrics treats it more like a secondary training set, leading to inflated metrics and poor generalization.

For example, techniques like extensively tuning hyperparameters or adding data augmentations targeted to boost validation accuracy can cause the model to fit too closely to the validation data. The model may achieve 99% validation accuracy but only 55% test accuracy.

Similarly, reusing the validation set for early stopping can also optimize the model specifically for that data. Stopping at the best validation performance overfits noise and fluctuations caused by the small validation size.

The validation set serves as a proxy to tune and select models. However, the goal remains maximizing real-world data performance, not the validation set. Minimizing the loss or error on validation data does not automatically translate to good generalization.

A good approach is to keep the use of the validation set minimal—hyperparameters can be tuned coarsely first on training data, for example. The validation set guides the training but should not influence or alter the model itself. It is a diagnostic, not an optimization tool.

When assessing performance on the validation set, care should be taken not to overfit. Tradeoffs are needed to build models that perform well on the overall population and are not overly tuned to the validation samples.

7.5 Optimization Algorithms

Stochastic gradient descent (SGD) is a simple yet powerful optimization algorithm for training machine learning models. It works by estimating the gradient of the loss function concerning the model parameters using a single training example and then updating the parameters in the direction that reduces the loss.

While conceptually straightforward, SGD needs a few areas for improvement. First, choosing a proper learning rate can be difficult—too small, and progress is very slow; too large, and parameters may oscillate and fail to converge. Second, SGD treats all parameters equally and independently, which may not be ideal in all cases. Finally, vanilla SGD uses only first-order gradient information, which results in slow progress on ill-conditioned problems.

7.5.1 Optimizations

Over the years, various optimizations have been proposed to accelerate and improve vanilla SGD. Ruder (2016) gives an excellent overview of the different optimizers. Briefly, several commonly used SGD optimization techniques include:

Momentum: Accumulates a velocity vector in directions of persistent gradient across iterations. This helps accelerate progress by dampening oscillations and maintains progress in consistent directions.

Nesterov Accelerated Gradient (NAG): A variant of momentum that computes gradients at the "look ahead" rather than the current parameter position. This anticipatory update prevents overshooting while the momentum maintains the accelerated progress.

Adagrad: An adaptive learning rate algorithm that maintains a per-parameter learning rate scaled down proportionate to each parameter's historical sum of gradients. This helps eliminate the need to tune learning rates (Duchi, Hazan, and Singer 2010) manually.

Adadelta: A modification to Adagrad restricts the window of accumulated past gradients, thus reducing the aggressive decay of learning rates (Zeiler 2012).

RMSProp: Divides the learning rate by an exponentially decaying average of squared gradients. This has a similar normalizing effect as Adagrad but does not accumulate the gradients over time, avoiding a rapid decay of learning rates (Hinton 2017).

Adam: Combination of momentum and rmsprop where rmsprop modifies the learning rate based on the average of recent magnitudes of gradients. Displays very fast initial progress and automatically tunes step sizes (Kingma and Ba 2014).

AMSGrad: A variant of Adam that ensures stable convergence by maintaining the maximum of past squared gradients, preventing the learning rate from increasing during training (S. J. Reddi, Kale, and Kumar 2019).

Of these methods, Adam has widely considered the go-to optimization algorithm for many deep-learning tasks. It consistently outperforms vanilla SGD in terms of training speed and performance. Other optimizers may be better suited in some cases, particularly for simpler models.

7.5.2 Tradeoffs

Table 7.2 is a pros and cons table for some of the main optimization algorithms for neural network training:

Table 7.2: Comparing the pros and cons of different optimization algorithms.

Algorithm	Pros	Cons
Momentum	• Faster convergence due to acceleration along gradients • Less oscillation than vanilla SGD	• Requires tuning of momentum parameter
Nesterov Accelerated Gradient (NAG)	• Faster than standard momentum in some cases • Anticipatory updates prevent overshooting	• More complex to understand intuitively
Adagrad	• Eliminates need to tune learning rates manually • Performs well on sparse gradients	• Learning rate may decay too quickly on dense gradients
Adadelta	• Less aggressive learning rate decay than Adagrad	• Still sensitive to initial learning rate value
RMSProp	• Automatically adjusts learning rates • Works well in practice	• No major downsides
Adam	• Combination of momentum and adaptive learning rates • Efficient and fast convergence	• Slightly worse generalization performance in some cases
AMSGrad	• Improvement to Adam addressing generalization issue	• Not as extensively used/tested as Adam

7.5.3 Benchmarking Algorithms

No single method is best for all problem types. This means we need comprehensive benchmarking to identify the most effective optimizer for specific datasets and models. The performance of algorithms like Adam, RMSProp, and Momentum varies due to batch size, learning rate schedules, model architecture, data distribution, and regularization. These variations underline the importance of evaluating each optimizer under diverse conditions.

Take Adam, for example, who often excels in computer vision tasks, unlike RMSProp, who may show better generalization in certain natural language processing tasks. Momentum's strength lies in its acceleration in scenarios with consistent gradient directions, whereas Adagrad's adaptive learning rates are more suited for sparse gradient problems.

This wide array of interactions among optimizers demonstrates the challenge of declaring a single, universally superior algorithm. Each optimizer has unique strengths, making it crucial to evaluate various methods to discover their optimal application conditions empirically.

A comprehensive benchmarking approach should assess the speed of convergence and factors like generalization error, stability, hyperparameter sensitivity, and computational efficiency, among others. This entails monitoring training and validation learning curves across multiple runs and comparing optimizers on various datasets and models to understand their strengths and weaknesses.

AlgoPerf, introduced by Dürr et al. (2021), addresses the need for a robust benchmarking system. This platform evaluates optimizer performance using criteria such as training loss curves, generalization error, sensitivity to hyperparameters, and computational efficiency. AlgoPerf tests various optimization methods, including Adam, LAMB, and Adafactor, across different model types like CNNs and RNNs/LSTMs on established datasets. It utilizes containerization and automatic metric collection to minimize inconsistencies and allows for controlled experiments across thousands of configurations, providing a reliable basis for comparing optimizers.

The insights gained from AlgoPerf and similar benchmarks are invaluable for guiding optimizers' optimal choice or tuning. By enabling reproducible evaluations, these benchmarks contribute to a deeper understanding of each optimizer's performance, paving the way for future innovations and accelerated progress in the field.

7.6 Hyperparameter Tuning

Hyperparameters are important settings in machine learning models that greatly impact how well your models ultimately perform. Unlike other model parameters that are learned during training, hyperparameters are specified by the data scientists or machine learning engineers before training the model.

Choosing the right hyperparameter values enables your models to learn patterns from data effectively. Some examples of key hyperparameters across ML algorithms include:

- **Neural networks:** Learning rate, batch size, number of hidden units, activation functions
- **Support vector machines:** Regularization strength, kernel type and parameters
- **Random forests:** Number of trees, tree depth
- **K-means:** Number of clusters

The problem is that there are no reliable rules of thumb for choosing optimal hyperparameter configurations—you typically have to try out different values and evaluate performance. This process is called hyperparameter tuning.

In the early years of modern deep learning, researchers were still grappling with unstable and slow convergence issues. Common pain points included training losses fluctuating wildly, gradients exploding or vanishing, and extensive trial-and-error needed to train networks reliably. As a result, an early focal point was using hyperparameters to control model optimization. For instance, seminal techniques like batch normalization allowed faster model convergence by tuning aspects of internal covariate shift. Adaptive learning rate methods also mitigated the need for extensive manual schedules. These addressed optimization issues during training, such as uncontrolled gradient divergence. Carefully adapted learning rates are also the primary control factor for achieving rapid and stable convergence even today.

As computational capacity expanded exponentially in subsequent years, much larger models could be trained without falling prey to pure numerical optimization issues. The focus shifted towards generalization - though efficient convergence was a core prerequisite. State-of-the-art techniques like Transformers brought in parameters in billions. At such sizes, hyperparameters around capacity, regularization, ensembling, etc., took center stage for tuning rather than only raw convergence metrics.

The lesson is that understanding the acceleration and stability of the optimization process itself constitutes the groundwork. Initialization schemes, batch sizes, weight decays, and other training hyperparameters remain indispensable today. Mastering fast and flawless convergence allows practitioners to expand their focus on emerging needs around tuning for metrics like accuracy, robustness, and efficiency at scale.

7.6.1 Search Algorithms

When it comes to the critical process of hyperparameter tuning, there are several sophisticated algorithms that machine learning practitioners rely on to search through the vast space of possible model configurations systematically. Some of the most prominent hyperparameter search algorithms include:

- **Grid Search:** The most basic search method, where you manually define a grid of values to check for each hyperparameter. For example, checking `learning rates = [0.01, 0.1, 1]` and batch `sizes = [32, 64, 128]`. The key advantage is simplicity, but it can lead to an exponential explosion in search space, making it time-consuming. It's best suited for fine-tuning a small number of parameters.

- **Random Search:** Instead of defining a grid, you randomly select values for each hyperparameter from a predefined range or set. This method is more efficient at exploring a vast hyperparameter space because it doesn't require an exhaustive search. However, it may still miss optimal parameters since it doesn't systematically explore all possible combinations.

- **Bayesian Optimization:** This is an advanced probabilistic approach for adaptive exploration based on a surrogate function to model performance over iterations. It is simple and efficient—it finds highly optimized hyperparameters in fewer evaluation steps. However, it requires more investment in setup (Snoek, Larochelle, and Adams 2012).

- **Evolutionary Algorithms:** These algorithms mimic natural selection principles. They generate populations of hyperparameter combinations and evolve them over time-based on performance. These algorithms offer robust search capabilities better suited for complex response surfaces. However, many iterations are required for reasonable convergence.

- **Population Based Training (PBT):** A method that optimizes hyperparameters by training multiple models in parallel, allowing them to share and adapt successful configurations during training, combining elements of random search and evolutionary algorithms (Jaderberg et al. 2017).

- **Neural Architecture Search:** An approach to designing well-performing architectures for neural networks. Traditionally, NAS approaches use some form of reinforcement learning to propose neural network architectures, which are then repeatedly evaluated (Zoph and Le 2016).

7.6.2 System Implications

Hyperparameter tuning can significantly impact time to convergence during model training, directly affecting overall runtime. The right values for key training hyperparameters are crucial for efficient model convergence. For example, the hyperparameter's learning rate controls the step size during gradient descent optimization. Setting a properly tuned learning rate schedule ensures the optimization algorithm converges quickly towards a good minimum. Too small a learning rate leads to painfully slow convergence, while too large a value causes the losses to fluctuate wildly. Proper tuning ensures rapid movement towards optimal weights and biases.

Similarly, the batch size for stochastic gradient descent impacts convergence stability. The right batch size smooths out fluctuations in parameter updates to approach the minimum faster. More batch sizes are needed to avoid noisy convergence, while large batch sizes fail to generalize and slow down convergence due to less frequent parameter updates. Tuning hyperparameters for faster convergence and reduced training duration has direct implications on cost and resource requirements for scaling machine learning systems:

- **Lower computational costs:** Shorter time to convergence means lower computational costs for training models. ML training often leverages large cloud computing instances like GPU and TPU clusters that incur heavy hourly charges. Minimizing training time directly reduces this resource rental cost, which tends to dominate ML budgets for organizations. Quicker iteration also lets data scientists experiment more freely within the same budget.

- **Reduced training time:** Reduced training time unlocks opportunities to train more models using the same computational budget. Optimized hyperparameters stretch available resources further, allowing businesses to develop and experiment with more models under resource constraints to maximize performance.

- **Resource efficiency:** Quicker training allows allocating smaller compute instances in the cloud since models require access to the resources for a shorter duration. For example, a one-hour training job allows using less powerful GPU instances compared to multi-hour training, which requires sustained compute access over longer intervals. This achieves cost savings, especially for large workloads.

There are other benefits as well. For instance, faster convergence reduces pressure on ML engineering teams regarding provisioning training resources. Simple model retraining routines can use lower-powered resources instead of requesting access to high-priority queues for constrained production-grade GPU clusters, freeing up deployment resources for other applications.

7.6.3 Auto Tuners

Given its importance, there is a wide array of commercial offerings to help with hyperparameter tuning. We will briefly touch on two examples: one focused on optimization for cloud-scale ML and the other for machine learning models targeting microcontrollers. Table 7.3 outlines the key differences:

Table 7.3: Comparison of optimization platforms for different machine learning use cases.

Platform	Target Use Case	Optimization Techniques	Benefits
Google's Vertex AI	Cloud-scale machine learning	Bayesian optimization, Population-Based training	Hides complexity, enabling fast, deployment-ready models with state-of-the-art hyperparameter optimization
Edge Impulse's EON Tuner	Microcontroller (TinyML) models	Bayesian optimization	Tailors models for resource-constrained devices, simplifies optimization for embedded deployment

7.6.3.1 BigML

Several commercial auto-tuning platforms are available to address this problem. One solution is Google's Vertex AI Cloud, which has extensive integrated support for state-of-the-art tuning techniques.

One of the most salient capabilities of Google's Vertex AI-managed machine learning platform is efficient, integrated hyperparameter tuning for model development. Successfully training performant ML models requires identifying optimal configurations for a set of external hyperparameters that dictate model behavior, posing a challenging high-dimensional search problem. Vertex AI simplifies this through Automated Machine Learning (AutoML) tooling.

Specifically, data scientists can leverage Vertex AI's hyperparameter tuning engines by providing a labeled dataset and choosing a model type such as a Neural Network or Random Forest classifier. Vertex launches a Hyperparameter Search job transparently on the backend, fully handling resource provisioning, model training, metric tracking, and result analysis automatically using advanced optimization algorithms.

Under the hood, Vertex AutoML employs various search strategies to intelligently explore the most promising hyperparameter configurations based on previous evaluation results. Among these, Bayesian Optimization is offered as it provides superior sample efficiency, requiring fewer training iterations to achieve optimized model quality compared to standard Grid Search or Random Search methods. For more complex neural architecture search spaces, Vertex AutoML utilizes Population-Based Training, which simultaneously trains multiple models and dynamically adjusts their hyperparameters by leveraging the performance of other models in the population, analogous to natural selection principles.

Vertex AI democratizes state-of-the-art hyperparameter search techniques at the cloud scale for all ML developers, abstracting away the underlying orchestration and execution complexity. Users focus solely on their dataset, model requirements, and accuracy goals, while Vertex manages the tuning cycle, resource allocation, model training, accuracy tracking, and artifact storage under the hood. The result is getting deployment-ready, optimized ML models faster for the target problem.

7.6.3.2 TinyML

Edge Impulse's Efficient On-device Neural Network Tuner (EON Tuner) is an automated hyperparameter optimization tool designed to develop microcontroller machine learning models. It streamlines the model development process by automatically finding the best neural network configuration for efficient and accurate deployment on resource-constrained devices.

The key functionality of the EON Tuner is as follows. First, developers define the model hyperparameters, such as number of layers, nodes per layer, activation functions, and learning rate annealing schedule. These parameters constitute the search space that will be optimized. Next, the target microcontroller platform is selected, providing embedded hardware constraints. The user can also specify optimization objectives, such as minimizing memory footprint, lowering latency, reducing power consumption, or maximizing accuracy.

With the defined search space and optimization goals, the EON Tuner leverages Bayesian hyperparameter optimization to explore possible configurations intelligently. Each prospective configuration is automatically implemented as a full model specification, trained, and evaluated for quality metrics. The continual process balances exploration and exploitation to arrive at optimized settings tailored to the developer's chosen chip architecture and performance requirements.

The EON Tuner frees machine learning engineers from the demandingly iterative process of hand-tuning models by automatically tuning models for embedded deployment. The tool integrates seamlessly into the Edge Impulse workflow, taking models from concept to efficiently optimized implementations on microcontrollers. The expertise encapsulated in EON Tuner regarding ML model optimization for microcontrollers ensures beginner and experienced developers alike can rapidly iterate to models fitting their project needs.

Caution 13: Hyperparameter Tuning

Get ready to unlock the secrets of hyperparameter tuning and take your PyTorch models to the next level! Hyperparameters are like the hidden dials and knobs that control your model's learning superpowers. In this Colab notebook, you'll team up with Ray Tune to find those perfect hyperparameter combinations. Learn how to define what values to search through, set up your training code for optimization, and let Ray Tune do the heavy lifting. By the end, you'll be a hyperparameter tuning pro!

Open in Colab

Video 7 explains the systematic organization of the hyperparameter tuning process.

! Important 7: Hyperparameter

https://www.youtube.com/watch?v=AXDByU3D1hA&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc&index=24

7.7 Regularization

Regularization is a critical technique for improving the performance and generalizability of machine learning models in applied settings. It refers to mathematically constraining or penalizing model complexity to avoid overfitting the training data. Without regularization, complex ML models are prone to overfitting the dataset and memorizing peculiarities and noise in the training set rather than learning meaningful patterns. They may achieve high training accuracy but perform poorly when evaluating new unseen inputs.

Regularization helps address this problem by placing constraints that favor simpler, more generalizable models that don't latch onto sampling errors. Techniques like L1/L2 regularization directly penalize large parameter values during training, forcing the model to use the smallest parameters that can adequately explain the signal. Early stopping rules halt training when validation set performance stops improving - before the model starts overfitting.

Appropriate regularization is crucial when deploying models to new user populations and environments where distribution shifts are likely. For example, an irregularized fraud detection model trained at a bank may work initially but accrue technical debt over time as new fraud patterns emerge.

Regularizing complex neural networks also offers computational advantages—smaller models require less data augmentation, compute power, and data storage. Regularization also allows for more efficient AI systems, where accuracy, robustness, and resource management are thoughtfully balanced against training set limitations.

Several powerful regularization techniques are commonly used to improve model generalization. Architecting the optimal strategy requires understanding how each method affects model learning and complexity.

7.7.1 L1 and L2

Two of the most widely used regularization forms are L1 and L2 regularization. Both penalize model complexity by adding an extra term to the cost function optimized during training. This term grows larger as model parameters increase.

L2 regularization, also known as ridge regression, adds the sum of squared magnitudes of all parameters multiplied by a coefficient α. This quadratic penalty curtails extreme parameter values more aggressively than L1 techniques. Implementation requires only changing the cost function and tuning α.

$$R_{L2}(\Theta) = \alpha \sum_{i=1}^{n} \theta_i^2$$

Where:

- $R_{L2}(\Theta)$ - The L2 regularization term that is added to the cost function

- α - The L2 regularization hyperparameter that controls the strength of regularization
- θ_i - The ith model parameter
- n - The number of parameters in the model
- θ_i^2 - The square of each parameter

And the full L2 regularized cost function is:

$$J(\theta) = L(\theta) + R_{L2}(\Theta)$$

Where:

- $L(\theta)$ - The original unregularized cost function
- $J(\theta)$ - The new regularized cost function

Both L1 and L2 regularization penalize large weights in the neural network. However, the key difference between L1 and L2 regularization is that L2 regularization penalizes the squares of the parameters rather than the absolute values. This key difference has a considerable impact on the resulting regularized weights. L1 regularization, or lasso regression, utilizes the absolute sum of magnitudes rather than the square multiplied by α. Penalizing the absolute value of weights induces sparsity since the gradient of the errors extrapolates linearly as the weight terms tend towards zero; this is unlike penalizing the squared value of the weights, where the penalty reduces as the weights tend towards 0. By inducing sparsity in the parameter vector, L1 regularization automatically performs feature selection, setting the weights of irrelevant features to zero. Unlike L2 regularization, L1 regularization leads to sparsity as weights are set to 0; in L2 regularization, weights are set to a value very close to 0 but generally never reach exact 0. L1 regularization encourages sparsity and has been used in some works to train sparse networks that may be more hardware efficient (Hoefler et al. 2021).

$$R_{L1}(\Theta) = \alpha \sum_{i=1}^{n} ||\theta_i||$$

Where:

- $R_{L1}(\Theta)$ - The L1 regularization term that is added to the cost function
- α - The L1 regularization hyperparameter that controls the strength of regularization
- θ_i - The i-th model parameter

- n - The number of parameters in the model
- $||\theta_i||$ - The L1 norm, which takes the absolute value of each parameter

And the full L1 regularized cost function is:

$$J(\theta) = L(\theta) + R_{L1}(\Theta)$$

Where:

- $L(\theta)$ - The original unregularized cost function
- $J(\theta)$ - The new regularized cost function

The choice between L1 and L2 depends on the expected model complexity and whether intrinsic feature selection is needed. Both require iterative tuning across a validation set to select the optimal α hyperparameter.

Video 8 and Video 9 explains how regularization works.

! Important 8: Regularization

https://www.youtube.com/watch?v=6g0t3Phly2M&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc&index=4

Video 9 explains how regularization can help reduce model overfitting to improve performance.

! Important 9: Why Regularization Reduces Overfitting

https://www.youtube.com/watch?v=NyG-7nRpsW8&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc&index=5

7.7.2 Dropout

Another widely adopted regularization method is dropout (Srivastava et al. 2014). During training, dropout randomly sets a fraction p of node outputs or hidden activations to zero. This encourages greater information distribution across more nodes rather than reliance on a small number of nodes. Come prediction time; the full neural network is used, with intermediate activations scaled by $1 - p$ to maintain output magnitudes. GPU optimizations make implementing dropout efficiently straightforward via frameworks like PyTorch and TensorFlow.

Let's be more pedantic. During training with dropout, each node's output a_i is passed through a dropout mask r_i before being used by the next layer:

$${}_i = r_i \odot a_i$$

Where:

- a_i - output of node i
- ${}_i$ - output of node i after dropout
- r_i - independent Bernoulli random variable with probability $1-p$ of being 1
- $\odot$ - elementwise multiplication

To understand how dropout works, it's important to know that the dropout mask r_i is based on Bernoulli random variables. A Bernoulli random variable takes a value of 1 with probability $1-p$ (keeping the activation) and a value of 0 with probability p (dropping the activation). This means that each node's activation is independently either kept or dropped during training. This dropout mask r_i randomly sets a fraction p of activations to 0 during training, forcing the network to make redundant representations.

At test time, the dropout mask is removed, and the activations are rescaled by $1-p$ to maintain expected output magnitudes:

$$a_i^{test} = (1-p)a_i$$

Where:

- a_i^{test} - node output at test time
- p - the probability of dropping a node.

The key hyperparameter is p, the probability of dropping each node,, often set between 0.2 and 0.5. Larger networks tend to benefit from more dropout, while small networks risk underfitting if too many nodes are cut out. Trial and error combined with monitoring validation performance helps tune the dropout level.

Video 10 discusses the intuition behind the dropout regularization technique and how it works.

! Important 10: Dropout

https://www.youtube.com/watch?v=ARq74QuavAo&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc&index=7

7.7.3 Early Stopping

The intuition behind early stopping involves tracking model performance on a held-out validation set across training epochs. At first, increases in training set fitness accompany gains in validation accuracy as the model picks up generalizable patterns. After some point, however, the model starts overfitting - latching onto peculiarities and noise in the training data that don't apply more broadly. The validation performance peaks and then degrades if training continues. Early stopping rules halt training at this peak to prevent overfitting. This technique demonstrates how ML pipelines must monitor system feedback, not just unquestioningly maximize performance on a static training set. The system's state evolves, and the optimal endpoints change.

Therefore, formal early stopping methods require monitoring a metric like validation accuracy or loss after each epoch. Common curves exhibit rapid initial gains that taper off, eventually plateauing and decreasing slightly as overfitting occurs. The optimal stopping point is often between 5 and 15 epochs past the peak, depending on patient thresholds. Tracking multiple metrics can improve signal since variance exists between measures.

Simple, early-stopping rules stop immediately at the first post-peak degradation. More robust methods introduce a patience parameter—the number of degrading epochs permitted before stopping. This avoids prematurely halting training due to transient fluctuations. Typical patience windows range from 50 to 200 validation batches. Wider windows incur the risk of overfitting. Formal tuning strategies can determine optimal patience.

Caution 14: Regularization

Battling Overfitting: Unlock the Secrets of Regularization! Overfitting is like your model memorizing the answers to a practice test, then failing the real exam. Regularization techniques are the study guides that help your model generalize and ace new challenges. In this Colab notebook, you'll learn how to tune regularization parameters for optimal results using L1 & L2 regularization, dropout, and early stopping.

Open in Colab

Video 11 covers a few other regularization methods that can reduce model overfitting.

! Important 11: Other Regularization Methods

https://www.youtube.com/watch?v=BOCLq2gpcGU&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc&index=8

7.8 Activation Functions

Activation functions play a crucial role in neural networks. They introduce nonlinear behaviors that allow neural nets to model complex patterns. Element-wise activation functions are applied to the weighted sums coming into each neuron in the network. Without activation functions, neural nets would be reduced to linear regression models.

Ideally, activation functions possess certain desirable qualities:

- **Nonlinear:** They enable modeling complex relationships through nonlinear transformations of the input sum.
- **Differentiable:** They must have well-defined first derivatives to enable backpropagation and gradient-based optimization during training.
- **Range-bounding:** They constrain the output signal, preventing an explosion. For example, sigmoid squashes inputs to (0,1).

Additionally, properties like computational efficiency, monotonicity, and smoothness make some activations better suited over others based on network architecture and problem complexity.

We will briefly survey some of the most widely adopted activation functions and their strengths and limitations. We will also provide guidelines for selecting appropriate functions matched to ML system constraints and use case needs.

7.8.1 Sigmoid

The sigmoid activation applies a squashing S-shaped curve tightly binding the output between 0 and 1. It has the mathematical form:

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

The exponentiation transform allows the function to smoothly transition from near 0 towards near 1 as the input moves from very negative to very positive. The monotonic rise covers the full (0,1) range.

The sigmoid function has several advantages. It always provides a smooth gradient for backpropagation, and its output is bounded between 0 and 1, which helps prevent "exploding" values during training. Additionally, it has a simple mathematical formula that is easy to compute.

However, the sigmoid function also has some drawbacks. It tends to saturate at extreme input values, which can cause gradients to "vanish," slowing down or even stopping the learning process. Furthermore, the function is not zero-centered, meaning that its outputs are not symmetrically distributed around zero, which can lead to inefficient updates during training.

7.8.2 Tanh

Tanh or hyperbolic tangent also assumes an S-shape but is zero-centered, meaning the average output value is 0.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The numerator/denominator transform shifts the range from (0,1) in Sigmoid to (-1, 1) in tanh.

Most pros/cons are shared with Sigmoid, but Tanh avoids some output saturation issues by being centered. However, it still suffers from vanishing gradients with many layers.

7.8.3 ReLU

The Rectified Linear Unit (ReLU) introduces a simple thresholding behavior with its mathematical form:

$$ReLU(x) = max(0, x)$$

It leaves all positive inputs unchanged while clipping all negative values to 0. This sparse activation and cheap computation make ReLU widely favored over sigmoid/tanh.

Figure 7.7 demonstrates the 3 activation functions we discussed above in comparison to a linear function:

7.8.4 Softmax

The softmax activation function is generally used as the last layer for classification tasks to normalize the activation value vector so that its elements sum to 1. This is useful for classification tasks where we want to learn to predict class-specific probabilities of a particular input, in

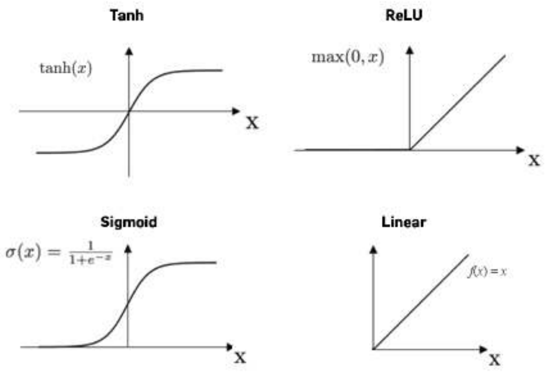


Figure 7.7: Common activation functions. Source: AI Wiki.

which case the cumulative probability across classes is equal to 1. The softmax activation function is defined as

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$$

7.8.5 Pros and Cons

Table 7.4 are the summarizing pros and cons of these various standard activation functions:

Table 7.4: Comparing the pros and cons of different optimization algorithms.

Activation	Pros	Cons
Sigmoid	• Smooth gradient for backdrop • Output bounded between 0 and 1	• Saturation kills gradients • Not zero-centered
Tanh	• Smoother gradient than sigmoid • Zero-centered output [-1, 1]	• Still suffers vanishing gradient issue
ReLU	• Computationally efficient • Introduces sparsity • Avoids vanishing gradients	• "Dying ReLU" units • Not bounded

Activation	Pros	Cons
Softmax	• Used for the last layer to normalize outputs to be a distribution • Typically used for classification tasks	

Caution 15: Activation Functions

Unlock the power of activation functions! These little mathematical workhorses are what make neural networks so incredibly flexible. In this Colab notebook, you'll go hands-on with functions like the Sigmoid, tanh, and the superstar ReLU. See how they transform inputs and learn which works best in different situations. It's the key to building neural networks that can tackle complex problems!

Open in Colab

7.9 Weight Initialization

Proper initialization of the weights in a neural network before training is a vital step directly impacting model performance. Randomly initializing weights to very large or small values can lead to problems like vanishing/exploding gradients, slow convergence of training, or getting trapped in poor local minima. Proper weight initialization accelerates model convergence during training and carries implications for system performance at inference time in production environments. Some key aspects include:

- **Faster Time-to-Accuracy:** Carefully tuned initialization leads to faster convergence, which results in models reaching target accuracy milestones earlier in the training cycle. For instance, Xavier initialization could reduce time-to-accuracy by 20% versus bad random initialization. As training is typically the most time- and compute-intensive phase, this directly enhances ML system velocity and productivity.

- **Model Iteration Cycle Efficiency:** If models train faster, the overall turnaround time for experimentation, evaluation, and model design iterations decreases significantly. Systems have more flexibility to explore architectures, data pipelines, etc, within given timeframes.

- **Impact on Necessary Training Epochs:** The training process runs for multiple epochs - with each full pass through the data being an epoch. Good Initialization can reduce the epochs required to converge the loss and accuracy curves on the training set by 10-30%. This means tangible resource and infrastructure cost savings.

- **Effect on Training Hyperparameters:** Weight initialization parameters interact strongly with certain regularization hyperparameters that govern the training dynamics, like learning rate schedules and dropout probabilities. Finding the right combination of settings is non-trivial. Appropriate Initialization smoothens this search.

Weight initialization has cascading benefits for machine learning engineering efficiency and minimized system resource overhead. It is an easily overlooked tactic that every practitioner should master. The choice of which weight initialization technique to use depends on factors like model architecture (number of layers, connectivity pattern, etc.), activation functions, and the specific problem being solved. Over the years, researchers have developed and empirically verified different initialization strategies targeted to common neural network architectures, which we will discuss here.

7.9.1 Uniform and Normal Initialization

When randomly initializing weights, two standard probability distributions are commonly used - uniform and Gaussian (normal). The uniform distribution sets an equal probability of the initial weight parameters falling anywhere within set minimum and maximum bounds. For example, the bounds could be -1 and 1, leading to a uniform spread of weights between these limits. The Gaussian distribution, on the other hand, concentrates probability around a mean value, following the shape of a bell curve. Most weight values will cluster in the region of the specified mean, with fewer samples towards the extreme ends. The standard deviation parameter controls the spread around the mean.

The choice between uniform or normal initialization depends on the network architecture and activation functions. For shallow networks, a normal distribution with a relatively small standard deviation (e.g., 0.01) is recommended. The bell curve prevents large weight values that could trigger training instability in small networks. For deeper networks, a normal distribution with higher standard deviation (say 0.5 or above) or uniform distribution may be preferred to account for vanishing gradient issues over many layers. The larger spread drives greater differentiation between neuron behaviors. Fine-tuning the initialization distribution parameters is crucial for stable and speedy model convergence. Monitoring training loss trends can diagnose issues for tweaking the parameters iteratively.

7.9.2 Xavier Initialization

Proposed by Glorot and Bengio (2010), this initialization technique is specially designed for sigmoid and tanh activation functions. These saturated activations can cause vanishing or exploding gradients during backpropagation over many layers.

The Xavier method cleverly sets the variance of the weight distribution based on the number of inputs and outputs to each layer. The intuition is that this balances the flow of information and gradients throughout the network. For example, consider a layer with 300 input units and 100 output units. Plugging this into the formula variance = 2/(#inputs + #outputs) gives a variance of 2/(300+100) = 0.01.

Sampling the initial weights from a uniform or normal distribution centered at 0 with this variance provides much smoother training convergence for deep sigmoid/tanh networks. The gradients are well-conditioned, preventing exponential vanishing or growth.

7.9.3 He Initialization

As proposed by K. He et al. (2015), this initialization technique is tailored to ReLU (Rectified Linear Unit) activation functions. ReLUs introduce the dying neuron problem where units get stuck outputting all 0s if they receive strong negative inputs initially. This slows and hinders training.

He overcomes this by sampling weights from a distribution with a variance set based only on the number of inputs per layer, disregarding the outputs. This keeps the incoming signals small enough to activate the ReLUs into their linear regime from the beginning, avoiding dead units. For a layer with 1024 inputs, the formula variance = 2/1024 = 0.002 keeps most weights concentrated closely around 0.

This specialized Initialization allows ReLU networks to converge efficiently right from the start. The choice between Xavier and He must match the intended network activation function.

Caution 16: Weight Initialization

Get your neural network off to a strong start with weight initialization! How you set those initial weights can make or break your model's training. Think of it like tuning the instruments in an orchestra before the concert. In this Colab notebook, you'll learn that the right initialization strategy can save time, improve model performance, and make your deep-learning journey much smoother.
Open in Colab

Video 12 emphasizes the importance of deliberately selecting initial weight values over random choices.

Important 12: Weight Initialization

https://www.youtube.com/watch?v=s2coXdufOzE&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc&index=11

7.10 System Bottlenecks

As introduced earlier, neural networks comprise linear operations (matrix multiplications) interleaved with element-wise nonlinear activation functions. The most computationally expensive portion of neural networks is the linear transformations, specifically the matrix multiplications between each layer. These linear layers map the activations from the previous layer to a higher dimensional space that serves as inputs to the next layer's activation function.

7.10.1 Runtime Complexity of Matrix Multiplication

7.10.1.1 Layer Multiplications vs. Activations

The bulk of computation in neural networks arises from the matrix multiplications between layers. Consider a neural network layer with an input dimension of $M = 500$ and output dimension of $N = 1000$; the matrix multiplication requires $O(N \cdot M) = O(1000 \cdot 500) = 500,000$ multiply-accumulate (MAC) operations between those layers.

Contrast this with the preceding layer, which had $M = 300$ inputs, requiring $O(500 \cdot 300) = 150,000$ ops. As the dimensions of the layers increase, the computational requirements scale quadratically with the size of the layer dimensions. The total computations across L layers can be expressed as $\sum_{l=1}^{L-1} O(N^{(l)} \cdot M^{(l-1)})$, where the computation required for each layer is dependent on the product of the input and output dimensions of the matrices being multiplied.

Now, comparing the matrix multiplication to the activation function, which requires only $O(N) = 1000$ element-wise nonlinearities for $N = 1000$ outputs, we can see the linear transformations dominating the activations computationally.

These large matrix multiplications impact hardware choices, inference latency, and power constraints for real-world neural network applications. For example, a typical DNN layer may require 500,000 multiply-accumulates vs. only 1000 nonlinear activations, demonstrating a 500x increase in mathematical operations.

When training neural networks, we typically use mini-batch gradient descent, operating on small batches of data simultaneously. Considering a batch size of B training examples, the input to the matrix multiplication becomes a $M \times B$ matrix, while the output is an $N \times B$ matrix.

7.10.1.2 Mini-batch

In training neural networks, we need to repeatedly estimate the gradient of the loss function with respect to the network parameters (i.e., weights, and biases). This gradient indicates which direction the parameters should be updated in to minimize the loss. As introduced previously, we perform updates over a batch of data points every update, also known as stochastic gradient descent or mini-batch gradient descent.

The most straightforward approach is to estimate the gradient based on a single training example, compute the parameter update, lather, rinse, and repeat for the next example. However, this involves very small and frequent parameter updates that can be computationally inefficient and may need to be more accurate in terms of convergence due to the stochasticity of using just a single data point for a model update.

Instead, mini-batch gradient descent balances convergence stability and computational efficiency. Rather than computing the gradient on single examples, we estimate the gradient based on small "minibatches" of data—usually between 8 and 256 examples in practice.

This provides a noisy but consistent gradient estimate that leads to more stable convergence. Additionally, the parameter update must only be performed once per mini-batch rather than once per example, reducing computational overhead.

By tuning the mini-batch size, we can control the tradeoff between the smoothness of the estimate (larger batches are generally better) and the frequency of updates (smaller batches allow more frequent updates). Mini-batch sizes are usually powers of 2, so they can efficiently leverage parallelism across GPU cores.

So, the total computation performs an $N \times M$ by $M \times B$ matrix multiplication, yielding $O(N \cdot M \cdot B)$ floating point operations. As a numerical example, $N = 1000$ hidden units, $M = 500$ input units, and a batch size $B = 64$ equates to 1000 x 500 x 64 = 32 million multiply-accumulates per training iteration!

In contrast, the activation functions are applied element-wise to the $N \times B$ output matrix, requiring only $O(N \cdot B)$ computations. For $N = 1000$ and $B = 64$, that is just 64,000 nonlinearities - 500X less work than the matrix multiplication.

As we increase the batch size to fully leverage parallel hardware like GPUs, the discrepancy between matrix multiplication and activation function cost grows even larger. This reveals how optimizing the linear algebra operations offers tremendous efficiency gains.

Therefore, matrix multiplication is central in analyzing where and how neural networks spend computation. For example, matrix multiplications often account for over 90% of inference latency and training time in common convolutional and recurrent neural networks.

7.10.1.3 Optimizing Matrix Multiplication

Several techniques improve the efficiency of general dense/sparse matrix-matrix and matrix-vector operations to improve overall efficiency. Some key methods include:

- Leveraging optimized math libraries like cuBLAS for GPU acceleration
- Enabling lower precision formats like FP16 or INT8 where accuracy permits
- Employing Tensor Processing Units with hardware matrix multiplication
- Sparsity-aware computations and data storage formats to exploit zero parameters
- Approximating matrix multiplications with algorithms like Fast Fourier Transforms
- Model architecture design to reduce layer widths and activations
- Quantization, pruning, distillation, and other compression techniques
- Parallelization of computation across available hardware

- Caching/pre-computing results where possible to reduce redundant operations

The potential optimization techniques are vast, given the outsized portion of time models spend in matrix and vector math. Even incremental improvements speed up runtimes and lower energy usage. Finding new ways to improve these linear algebra primitives remains an active area of research aligned with the future demands of machine learning. We will discuss these in detail in the Optimizations and AI Acceleration chapters.

7.10.2 Compute vs. Memory Bottleneck

At this point, matrix-matrix multiplication is the core mathematical operation underpinning neural networks. Both training and inference for neural networks heavily use these matrix multiply operations. Analysis shows that over 90% of computational requirements in state-of-the-art neural networks arise from matrix multiplications. Consequently, the performance of matrix multiplication has an enormous influence on overall model training or inference time.

7.10.2.1 Training versus Inference

While training and inference rely heavily on matrix multiplication performance, their precise computational profiles differ. Specifically, neural network inference tends to be more compute-bound than training for an equivalent batch size. The key difference lies in the backpropagation pass, which is only required during training. Backpropagation involves a sequence of matrix multiply operations to calculate gradients with respect to activations across each network layer. Critically, though, no additional memory bandwidth is needed here—the inputs, outputs, and gradients are read/written from cache or registers.

As a result, training exhibits lower arithmetic intensities, with gradient calculations bounded by memory access instead of **FLOPs** (Floating Point Operations Per Second), a measure of computational performance that indicates how many floating-point calculations a system can perform per second. In contrast, the forward propagation dominates neural network inference, which corresponds to a series of matrix-matrix multiplies. With no memory-intensive gradient retrospecting, larger batch sizes readily push inference into being extremely compute-bound. The high measured arithmetic intensities exhibit this. Response times may be critical for some inference applications, forcing the application provider to use a smaller batch size to meet these response-time requirements, thereby reducing hardware efficiency; hence, inferences may see lower hardware utilization.

The implications are that hardware provisioning and bandwidth vs. FLOP tradeoffs differ depending on whether a system targets training or inference. High-throughput, low-latency servers for inference should emphasize computational power instead of memory, while training clusters require a more balanced architecture.

However, matrix multiplication exhibits an interesting tension - the underlying hardware's memory bandwidth or arithmetic throughput capabilities can bind it. The system's ability to fetch and supply matrix data versus its ability to perform computational operations determines this direction.

This phenomenon has profound impacts; hardware must be designed judiciously, and software optimizations must be considered. Optimizing and balancing compute versus memory to alleviate this underlying matrix multiplication bottleneck is crucial for efficient model training and deployment.

Finally, batch size may impact convergence rates during neural network training, another important consideration. For example, there are generally diminishing returns in benefits to convergence with extremely large batch sizes (i.e.,> 16384). In contrast, extremely large batch sizes may be increasingly beneficial from a hardware/arithmetic intensity perspective; using such large batches may not translate to faster convergence vs wall-clock time due to their diminishing benefits to convergence. These tradeoffs are part of the design decisions core to systems for the machine-learning type of research.

7.10.2.2 Batch Size

The batch size used during neural network training and inference significantly impacts whether matrix multiplication poses more of a computational or memory bottleneck. Concretely, the batch size refers to the number of samples propagated through the network together in one forward/backward pass. Matrix multiplication equates to larger matrix sizes.

Specifically, let's look at the arithmetic intensity of matrix multiplication during neural network training. This measures the ratio between computational operations and memory transfers. The matrix multiply of two matrices of size $N \times M$ and $M \times B$ requires $N \times M \times B$ multiply-accumulate operations, but only transfers of $N \times M + M \times B$ matrix elements.

As we increase the batch size B, the number of arithmetic operations grows faster than the memory transfers. For example, with a batch size of 1, we need $N \times M$ operations and $N + M$ transfers, giving an arithmetic intensity ratio of around $\frac{N \times M}{N+M}$. But with a large batch size of 128, the intensity ratio becomes $\frac{128 \times N \times M}{N \times M + M \times 128} \approx 128$.

Using a larger batch size shifts the overall computation from memory-bounded to more compute-bounded. AI training uses large batch sizes and is generally limited by peak arithmetic computational performance, i.e., Application 3 in Figure 7.8. Therefore, batched matrix multiplication is far more computationally intensive than memory access bound. This has implications for hardware design and software optimizations, which we will cover next. The key insight is that we can significantly alter the computational profile and bottlenecks posed by neural network training and inference by tuning the batch size.

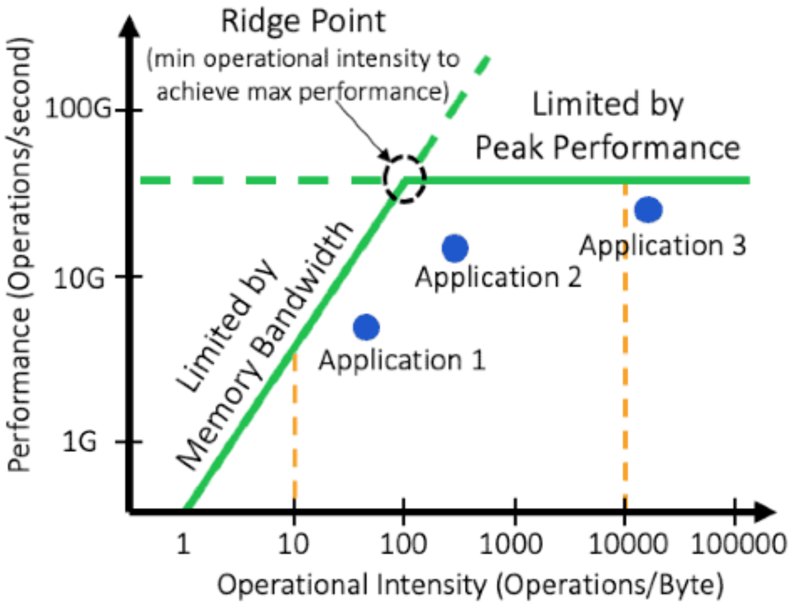


Figure 7.8: AI training roofline model.

7.10.2.3 Hardware Characteristics

Modern hardware like CPUs and GPUs is highly optimized for computational throughput rather than memory bandwidth. For example, high-end H100 Tensor Core GPUs can deliver over 60 TFLOPS of double-precision performance but only provide up to 3 TB/s of memory bandwidth. This means there is almost a 20x imbalance between arithmetic units and memory access; consequently, for hardware like

GPU accelerators, neural network training workloads must be made as computationally intensive as possible to use the available resources fully.

This further motivates the need for using large batch sizes during training. When using a small batch, the matrix multiplication is bounded by memory bandwidth, underutilizing the abundant compute resources. However, we can shift the bottleneck towards computation and attain much higher arithmetic intensity with sufficiently large batches. For instance, batches of 256 or 512 samples may be needed to saturate a high-end GPU. The downside is that larger batches provide less frequent parameter updates, which can impact convergence. Still, the parameter serves as an important tuning knob to balance memory vs compute limitations.

Therefore, given the imbalanced compute-memory architectures of modern hardware, employing large batch sizes is essential to alleviate bottlenecks and maximize throughput. As mentioned, the subsequent software and algorithms also need to accommodate such batch sizes since larger batch sizes may have diminishing returns toward the network's convergence. Using very small batch sizes may lead to suboptimal hardware utilization, ultimately limiting training efficiency. Scaling up to large batch sizes is a research topic explored in various works that aim to do large-scale training (Y. You et al. 2017).

7.10.2.4 Model Architectures

The underlying neural network architecture also affects whether matrix multiplication poses more of a computational or memory bottleneck during execution. Transformers and MLPs are much more compute-bound than CNN convolutional neural networks. This stems from the types of matrix multiplication operations involved in each model. Transformers rely on self-attention, multiplying large activation matrices by massive parameter matrices to relate elements. MLPs stack fully connected layers, also requiring large matrix multiplies.

In contrast, the convolutional layers in CNNs have a sliding window that reuses activations and parameters across the input, which means fewer unique matrix operations are needed. However, the convolutions require repeatedly accessing small input parts and moving partial sums to populate each window. Even though the arithmetic operations in convolutions are intense, this data movement and buffer manipulation impose huge memory access overheads. CNNs comprise several layered stages, so intermediate outputs must frequently materialize in memory.

As a result, CNN training tends to be more memory bandwidth bound relative to arithmetic bound compared to Transformers and

MLPs. Therefore, the matrix multiplication profile, and in turn, the bottleneck posed, varies significantly based on model choice. Hardware and systems need to be designed with appropriate compute-memory bandwidth balance depending on target model deployment. Models relying more on attention and MLP layers require higher arithmetic throughput compared to CNNs, which necessitates high memory bandwidth.

7.11 Training Parallelization

Training neural networks entails intensive computational and memory demands. The backpropagation algorithm for calculating gradients and updating weights consists of repeated matrix multiplications and arithmetic operations over the entire dataset. For example, one pass of backpropagation scales in time complexity with $O(num\_parameters \times batch\_size \times sequence\_length)$.

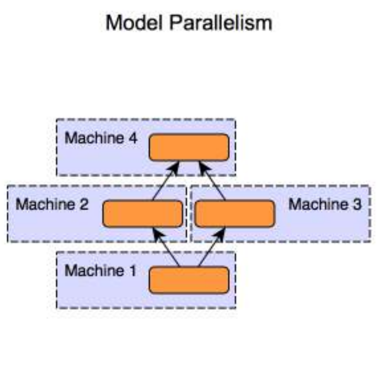
The computational requirements grow rapidly as model size increases in parameters and layers. Moreover, the algorithm requires storing activation outputs and model parameters for the backward pass, which grows with model size.

Larger models cannot fit and train on a single accelerator device like a GPU, and the memory footprint becomes prohibitive. Therefore, we need to parallelize model training across multiple devices to provide sufficient compute and memory to train state-of-the-art neural networks.

As shown in Figure 7.9, the two main approaches are data parallelism, which replicates the model across devices while splitting the input data batch-wise, and model parallelism, which partitions the model architecture itself across different devices. By training in parallel, we can leverage greater aggregate compute and memory resources to overcome system limitations and accelerate deep learning workloads.

7.11.1 Data Parallel

Data parallelization is a common approach to parallelize machine learning training across multiple processing units, such as GPUs or distributed computing resources. The training dataset is divided into batches in data parallelism, and a separate processing unit processes each batch. The model parameters are then updated based on the gradients computed from the processing of each batch. Here's a step-by-step description of data parallelization for ML training:



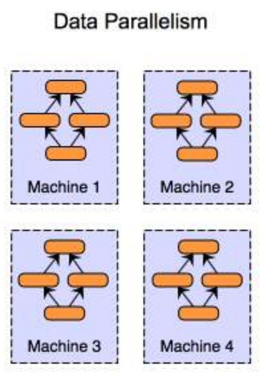


Figure 7.9: Data parallelism versus model parallelism.

1. **Dividing the Dataset:** The training dataset is divided into smaller batches, each containing a subset of the training examples.

2. **Replicating the Model:** The neural network model is replicated across all processing units, and each processing unit has its copy of the model.

3. **Parallel Computation:** Each processing unit takes a different batch and independently computes the forward and backward passes. During the forward pass, the model makes predictions on the input data. The loss function determines gradients for the model parameters during the backward pass.

4. **Gradient Aggregation:** After processing their respective batches, the gradients from each processing unit are aggregated. Common aggregation methods include summation or averaging of the gradients.

5. **Parameter Update:** The aggregated gradients update the model parameters. The update can be performed using optimization algorithms like SGD or variants like Adam.

6. **Synchronization:** After the update, all processing units synchronize their model parameters, ensuring that each has the latest version of the model.

The prior steps are repeated for several iterations or until convergence.

Let's take a specific example. We have 256 batch sizes and 8 GPUs; each GPU will get a micro-batch of 32 samples. Their forward and backward passes compute losses and gradients only based on the local 32 samples. The gradients get aggregated across devices with a parameter server or collective communications library to get the effective gradient for the global batch. Weight updates happen independently on each GPU according to these gradients. After a configured number of iterations, updated weights synchronize and equalize across devices before continuing to the next iterations.

Data parallelism is effective when the model is large, and the dataset is substantial, as it allows for parallel processing of different parts of the data. It is widely used in deep learning frameworks and libraries that support distributed training, such as TensorFlow and PyTorch. However, to ensure efficient parallelization, care must be taken to handle issues like communication overhead, load balancing, and synchronization.

7.11.2 Model Parallelism

Model parallelism refers to distributing the neural network model across multiple devices rather than replicating the full model like data parallelism. This is particularly useful when a model is too large to fit into the memory of a single GPU or accelerator device. While this might not be specifically applicable for embedded or TinyML use cases as most models are relatively small(er), it is still useful to know.

In model parallel training, different parts or layers of the model are assigned to separate devices. The input activations and intermediate outputs get partitioned and passed between these devices during the forward and backward passes to coordinate gradient computations across model partitions.

The memory footprint and computational operations are distributed by splitting the model architecture across multiple devices instead of concentrating on one. This enables training very large models with billions of parameters that otherwise exceed the capacity of a single device. There are several main ways in which we can do partitioning:

- **Layer-wise parallelism:** Consecutive layers are distributed onto different devices. For example, device 1 contains layers 1-3; device 2 contains layers 4-6. The output activations from layer 3 would be transferred to device 2 to start the next layers for the forward pass computations.

- **Filter-wise parallelism:** In convolutional layers, output filters can be split among devices. Each device computes activation outputs for a subset of filters, which get concatenated before propagating further.

- **Spatial parallelism:** The input images get divided spatially, so each device processes over a certain region like the top-left quarter of images. The output regions then combine to form the full output.

Additionally, hybrid combinations can split the model layer-wise and data batch-wise. The appropriate type of model parallelism depends on the specific neural architecture constraints and hardware setup. Optimizing the partitioning and communication for the model topology is key to minimizing overhead.

However, as the model parts run on physically separate devices, they must communicate and synchronize their parameters during each training step. The backward pass must ensure gradient updates propagate accurately across the model partitions. Hence, coordination and high-speed interconnecting between devices are crucial for optimizing the performance of model parallel training. Careful partitioning and communication protocols are required to minimize transfer overhead.

7.11.3 Comparison

To summarize, Table 7.5 demonstrates some of the key characteristics for comparing data parallelism and model parallelism.

Table 7.5: Comparing data parallelism and model parallelism.

Characteristic	Data Parallelism	Model Parallelism
Definition	Distribute data across devices with replicas	Distribute model across devices
Objective	Accelerate training through compute scaling	Enable larger model training
Scaling Method	Scale devices/workers	Scale model size
Main Constraint	Model size per device	Device coordination overhead
Hardware Requirements	Multiple GPU/TPUs	Often specialized interconnect
Primary Challenge	Parameter synchronization	Complex partitioning and communication
Types	N/A	Layer-wise, filter-wise, spatial

Characteristic	Data Parallelism	Model Parallelism
Code Complexity	Minimal changes	More significant model surgery
Popular Libraries	Horovod, PyTorch Distributed	Mesh TensorFlow

7.12 Conclusion

In this chapter, we have covered the core foundations that enable effective training of artificial intelligence models. We explored the mathematical concepts like loss functions, backpropagation, and gradient descent that make neural network optimization possible. We also discussed practical techniques around leveraging training data, regularization, hyperparameter tuning, weight initialization, and distributed parallelization strategies that improve convergence, generalization, and scalability.

These methodologies form the bedrock through which the success of deep learning has been attained over the past decade. Mastering these fundamentals equips practitioners to architect systems and refine models tailored to their problem context. However, as models and datasets grow exponentially, training systems must optimize across metrics like time, cost, and carbon footprint. Hardware scaling through warehouse scales enables massive computational throughput - but optimizations around efficiency and specialization will be key. Software techniques like compression and sparsity exploitation can increase hardware gains. We will discuss several of these in the coming chapters.

Overall, the fundamentals covered in this chapter equip practitioners to build, refine, and deploy models. However, interdisciplinary skills spanning theory, systems, and hardware will differentiate experts who can lift AI to the next level sustainably and responsibly that society requires. Understanding efficiency alongside accuracy constitutes the balanced engineering approach needed to train intelligent systems that integrate smoothly across many real-world contexts.

7.13 Resources

Here is a curated list of resources to support students and instructors in their learning and teaching journeys. We are continuously working on expanding this collection and will be adding new exercises soon.

Slides

These slides are a valuable tool for instructors to deliver lectures and for students to review the material at their own pace. We encourage students and instructors to leverage these slides to improve their understanding and facilitate effective knowledge transfer.

- Thinking About Loss.
- Minimizing Loss.
- Training, Validation, and Test Data.
- Continuous Training:
  - Retraining Trigger.
  - Data Processing Overview.
  - Data Ingestion.
  - Data Validation.
  - Data Transformation.
  - Training with AutoML.
  - Continuous Training with Transfer Learning.
  - Continuous Training Use Case Metrics.
  - Continuous Training Impact on MLOps.

Videos

- Video 5
- Video 6
- Video 7
- Video 8
- Video 9
- Video 10
- Video 11
- Video 12

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.

- Exercise 12
- Exercise 13
- Exercise 14
- Exercise 16
- Exercise 15

Labs

In addition to exercises, we offer a series of hands-on labs allowing students to gain practical experience with embedded AI technologies. These labs provide step-by-step guidance, enabling students to develop their skills in a structured and supportive environment. We are excited to announce that new labs will be available soon, further enriching the learning experience.

- *Coming soon.*

Chapter 8

Efficient AI

Figure 8.1: *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.*

Efficiency in artificial intelligence is not simply a luxury but a necessity. In this chapter, we dive into the key concepts underpinning AI systems' efficiency. 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. The pursuit of efficiency guarantees that AI models are streamlined, rapid, and sustainable, thereby widening their applicability across various platforms and scenarios.

Learning Objectives

- Recognize the need for efficient AI in TinyML/edge devices.
- Understand the need for efficient model architectures like MobileNets and SqueezeNet.
- Understand why techniques for model compression are important.
- Gain an appreciation for the value of efficient AI hardware.
- Recognize the importance of numerical representations and their precision.
- Understand the nuances of model comparison beyond just accuracy.
- Recognize that model comparison involves memory, computation, power, and speed, not just accuracy.
- Recognize efficiency encompasses technology, costs, and ethics.

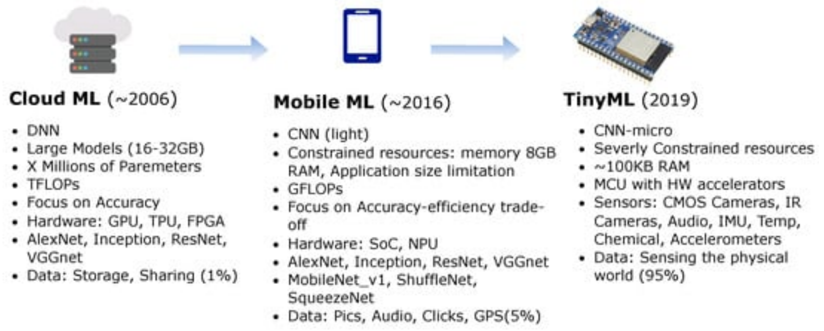
The focus is on gaining a conceptual understanding of the motivations and significance of the various strategies for achieving efficient AI, both in terms of techniques and a holistic perspective. Subsequent chapters provide a more in-depth exploration of these multiple concepts.

8.1 Overview

Training models can consume significant energy, sometimes equivalent to the carbon footprint of sizable industrial processes. We will cover some of these sustainability details in the AI Sustainability chapter. On the deployment side, if these models are not optimized for efficiency, they can quickly drain device batteries, demand excessive memory, or fall short of real-time processing needs. Through this chapter, we aim to elucidate the nuances of efficiency, setting the groundwork for a comprehensive exploration in the subsequent chapters.

8.2 The Need for Efficient AI

Efficiency takes on different connotations depending on where AI computations occur. Let's revisit Cloud, Edge, and TinyML (as discussed in ML Systems) and differentiate between them in terms of efficiency. Figure 8.2 provides a big-picture comparison of the three different platforms.



Cloud ML (~2006)
• DNN
• Large Models (16-32GB)
• X Millions of Paremeters
• TFLOPs
• Focus on Accuracy
• Hardware: GPU, TPU, FPGA
• AlexNet, Inception, ResNet, VGGnet
• Data: Storage, Sharing (1%)
Mobile ML (~2016)
• CNN (light)
• Constrained resources: memory 8GB RAM, Application size limitation
• GFLOPs
• Focus on Accuracy-efficiency trade-off
• Hardware: SoC, NPU
• AlexNet, Inception, ResNet, VGGnet
• MobileNet_v1, ShuffleNet, SqueezeNet
• Data: Pics, Audio, Clicks, GPS(5%)
TinyML (2019)
• CNN-micro
• Severly Constrained resources
• ~100KB RAM
• MCU with HW accelerators
• Sensors: CMOS Cameras, IR Cameras, Audio, IMU, Temp, Chemical, Accelerometers
• Data: Sensing the physical world (95%)

Figure 8.2: Cloud, Mobile and TinyML. Source: Schizas et al. (2022).

Cloud AI: Traditional AI models often run in large-scale data centers equipped with powerful GPUs and TPUs (Barroso, Hölzle, and Ranganathan 2019). Here, efficiency pertains to optimizing computational resources, reducing costs, and ensuring timely data processing and return. However, relying on the cloud introduces latency, especially when dealing with large data streams that require uploading, processing, and downloading.

Edge AI: Edge computing brings AI closer to the data source, processing information directly on local devices like smartphones, cameras, or industrial machines (E. Li et al. 2020). Here, efficiency encompasses quick real-time responses and reduced data transmission needs. However, the constraints are tighter—these devices, while more powerful than microcontrollers, have limited computational power compared to cloud setups.

TinyML: TinyML pushes the boundaries by enabling AI models to run on microcontrollers or extremely resource-constrained environments. The processor and memory performance difference between TinyML and cloud or mobile systems can be several orders of magnitude (Warden and Situnayake 2019). Efficiency in TinyML is about ensuring models are lightweight enough to fit on these devices, consume minimal energy (critical for battery-powered devices), and still perform their tasks effectively.

The spectrum from Cloud to TinyML represents a shift from vast, centralized computational resources to distributed, localized, and constrained environments. As we transition from one to the other, the challenges and strategies related to efficiency evolve, underlining the need for specialized approaches tailored to each scenario. Having established the need for efficient AI, especially within the context of TinyML, we will transition to exploring the methodologies devised to meet these challenges. The following sections outline the main concepts we will dive deeper into later. We will demonstrate the breadth and depth of innovation needed to achieve efficient AI as we explore these strategies.

8.3 Efficient Model Architectures

Selecting an optimal model architecture is as crucial as optimizing it. In recent years, researchers have made significant strides in exploring innovative architectures that can inherently have fewer parameters while maintaining strong performance.

MobileNets: MobileNets are efficient mobile and embedded vision application models (Howard et al. 2017). The key idea that led to their success is depth-wise separable convolutions, significantly reducing the number of parameters and computations in the network. MobileNetV2 and V3 further enhance this design by introducing inverted residuals and linear bottlenecks.

SqueezeNet: SqueezeNet is a class of ML models known for its smaller size without sacrificing accuracy. It achieves this by using a "fire module" that reduces the number of input channels to 3x3 filters, thus reducing the parameters (Iandola et al. 2016). Moreover, it employs delayed downsampling to increase the accuracy by maintaining a larger feature map.

ResNet variants: The Residual Network (ResNet) architecture allows for the introduction of skip connections or shortcuts (K. He et al. 2016). Some variants of ResNet are designed to be more efficient. For instance, ResNet-SE incorporates the "squeeze and excitation" mechanism to recalibrate feature maps (J. Hu, Shen, and Sun 2018), while ResNeXt offers grouped convolutions for efficiency (S. Xie et al. 2017).

8.4 Efficient Model Compression

Model compression methods are essential for bringing deep learning models to devices with limited resources. These techniques reduce models' size, energy consumption, and computational demands without significantly losing accuracy. At a high level, the methods can be categorized into the following fundamental methods:

Pruning: We've mentioned pruning a few times in previous chapters but have not yet formally introduced it. Pruning is similar to trimming the branches of a tree. This was first thought of in the Optimal Brain Damage paper (LeCun, Denker, and Solla 1989) and was later popularized in the context of deep learning by Han, Mao, and Dally (2016). Certain weights or entire neurons are removed from the network in pruning based on specific criteria. This can significantly reduce the model size. We will explore two of the main pruning strategies, structured and unstructured pruning, in Section 9.2.1. Figure 8.3 is an example of neural network pruning, where removing some of the nodes in the inner layers (based on specific criteria) reduces the number of edges between the nodes and, in turn, the model's size.

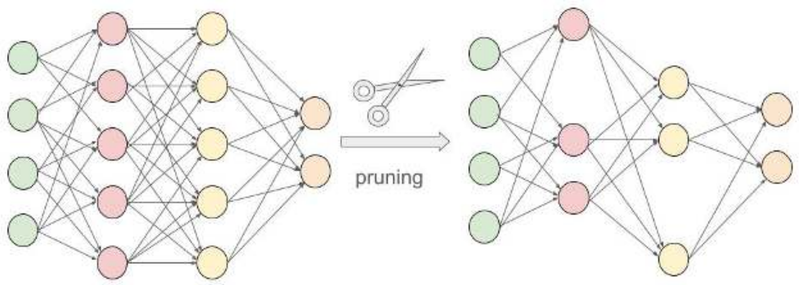


Figure 8.3: Neural Network Pruning.

Quantization: Quantization is the process of constraining an input from a large set to output in a smaller set, primarily in deep learning; this means reducing the number of bits that represent the weights and biases of the model. For example, using 16-bit or 8-bit representations instead of 32-bit can reduce the model size and speed up computations, with a minor trade-off in accuracy. We will explore these in more detail in Section 9.3.4. Figure 8.4 shows an example of quantization by rounding to the closest number. The conversion from 32-bit floating point to 16-bit reduces memory usage by 50%. Going from a 32-bit to an 8-bit integer reduces memory usage by 75%. While the loss in numeric precision, and consequently model performance, is minor, the memory usage efficiency is significant.

Knowledge Distillation: Knowledge distillation involves training a smaller model (student) to replicate the behavior of a larger model (teacher). The idea is to transfer the knowledge from the cumbersome

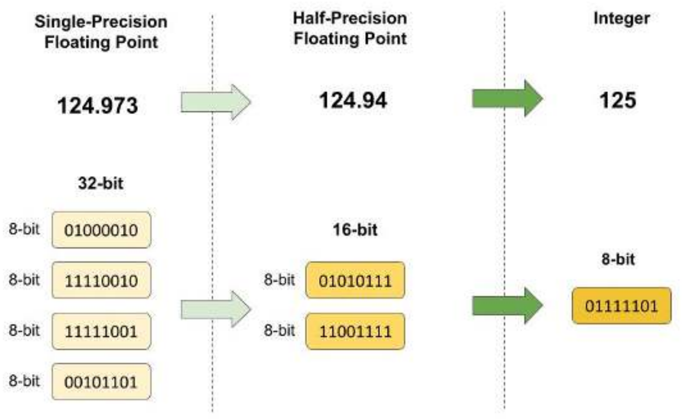


Figure 8.4: Different forms of quantization.

model to the lightweight one. Hence, the smaller model attains performance close to its larger counterpart but with significantly fewer parameters. Figure 8.5 demonstrates the tutor-student framework for knowledge distillation. We will explore knowledge distillation in more detail in the Section 9.2.2.1.

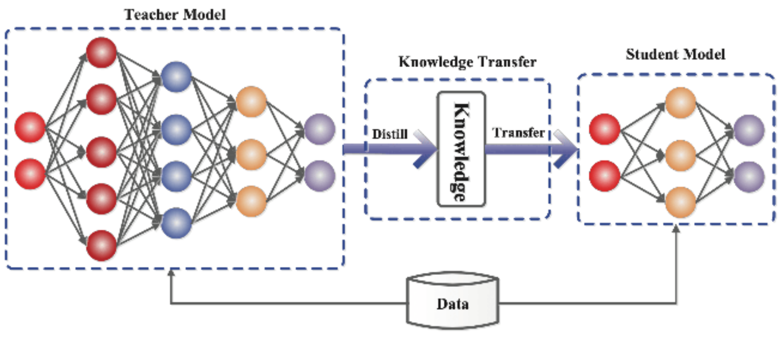


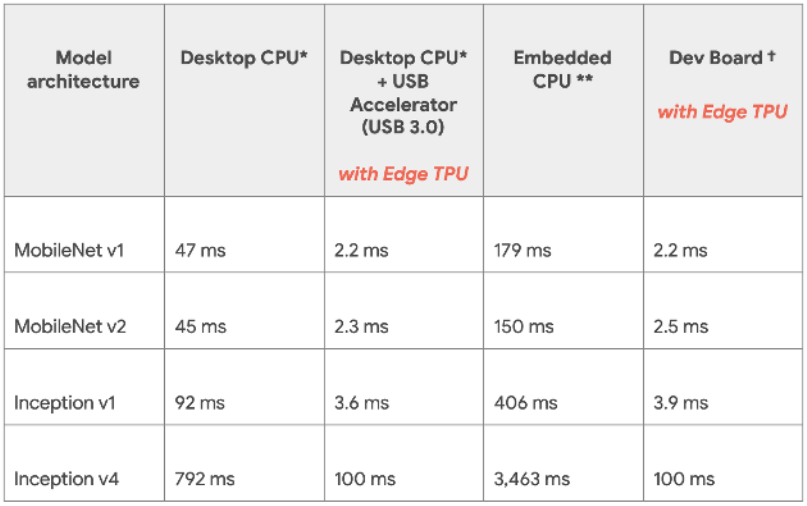
Figure 8.5: The tutor-student framework for knowledge distillation. Source: Medium

8.5 Efficient Inference Hardware

In the Training chapter, we discussed the process of training AI models. Now, from an efficiency standpoint, it's important to note that training is a resource and time-intensive task, often requiring powerful hardware and taking anywhere from hours to weeks to complete. Inference, on the other hand, needs to be as fast as possible, especially in real-time applications. This is where efficient inference hardware comes into play. By optimizing the hardware specifically for inference tasks, we can achieve rapid response times and power-efficient operation, which is especially crucial for edge devices and embedded systems.

TPUs (Tensor Processing Units): TPUs are custom-built ASICs (Application-Specific Integrated Circuits) by Google to accelerate machine learning workloads (N. P. Jouppi et al. 2017a). They are optimized for tensor operations, offering high throughput for low-precision arithmetic, and are designed specifically for neural network machine learning. TPUs significantly accelerate model training and inference compared to general-purpose GPU/CPUs. This boost means faster model training and real-time or near-real-time inference capabilities, crucial for applications like voice search and augmented reality.

Edge TPUs are a smaller, power-efficient version of Google's TPUs tailored for edge devices. They provide fast on-device ML inferencing for TensorFlow Lite models. Edge TPUs allow for low-latency, high-efficiency inference on edge devices like smartphones, IoT devices, and embedded systems. AI capabilities can be deployed in real-time applications without communicating with a central server, thus saving bandwidth and reducing latency. Consider the table in Figure 8.6. It shows the performance differences between running different models on CPUs versus a Coral USB accelerator. The Coral USB accelerator is an accessory by Google's Coral AI platform that lets developers connect Edge TPUs to Linux computers. Running inference on the Edge TPUs was 70 to 100 times faster than on CPUs.



Model architecture	Desktop CPU*	Desktop CPU* + USB Accelerator (USB 3.0) *with Edge TPU*	Embedded CPU **	Dev Board † *with Edge TPU*
MobileNet v1	47 ms	2.2 ms	179 ms	2.2 ms
MobileNet v2	45 ms	2.3 ms	150 ms	2.5 ms
Inception v1	92 ms	3.6 ms	406 ms	3.9 ms
Inception v4	792 ms	100 ms	3,463 ms	100 ms

Figure 8.6: Accelerator vs CPU performance comparison across different hardware configurations. Desktop CPU: 64-bit Intel(R) Xeon(R) E5-1650 v4 @ 3.60GHz. Embedded CPU: Quad-core Cortex-A53 @ 1.5GHz, †Dev Board: Quad-core Cortex-A53 @ 1.5GHz + Edge TPU. Source: TensorFlow Blog.

NN (Neural Network) Accelerators: Fixed-function neural network accelerators are hardware accelerators designed explicitly for neural network computations. They can be standalone chips or part of a larger system-on-chip (SoC) solution. By optimizing the hardware for the specific operations that neural networks require, such as matrix multiplications and convolutions, NN accelerators can achieve faster inference times and lower power consumption than general-purpose CPUs and GPUs. They are especially beneficial in TinyML devices with power or thermal constraints, such as smartwatches, micro-drones, or robotics.

But these are all but the most common examples. Several other types of hardware are emerging that have the potential to offer significant advantages for inference. These include, but are not limited to, neuromorphic hardware, photonic computing, etc. In Section 10.3, we will explore these in greater detail.

Efficient hardware for inference speeds up the process, saves energy, extends battery life, and can operate in real-time conditions. As AI continues to be integrated into myriad applications, from smart cameras to voice assistants, the role of optimized hardware will only become more prominent. By leveraging these specialized hardware components, developers and engineers can bring the power of AI to devices and situations that were previously unthinkable.

8.6 Efficient Numerics

Machine learning, and especially deep learning, involves enormous amounts of computation. Models can have millions to billions of parameters, often trained on vast datasets. Every operation, every multiplication or addition, demands computational resources. Therefore, the precision of the numbers used in these operations can significantly impact the computational speed, energy consumption, and memory requirements. This is where the concept of efficient numerics comes into play.

8.6.1 Numerical Formats

There are many different types of numerics. Numerics have a long history in computing systems.

Floating point: Known as a single-precision floating point, FP32 utilizes 32 bits to represent a number, incorporating its sign, exponent, and mantissa. Understanding how floating point numbers are represented under the hood is crucial for grasping the various optimizations possible in numerical computations. The sign bit determines whether the number is positive or negative, the exponent controls the range of values that can be represented, and the mantissa determines the precision of the number. The combination of these components allows floating point numbers to represent a vast range of values with varying degrees of precision.

Video 13 provides a comprehensive overview of these three main components - sign, exponent, and mantissa - and how they work together to represent floating point numbers.

! Important 13: Floating Point Numbers

https://youtu.be/gc1Nl3mmCuY?si=nImcymfbE5H392vu

FP32 is widely adopted in many deep learning frameworks and balances accuracy and computational requirements. It is prevalent in the training phase for many neural networks due to its sufficient precision in capturing minute details during weight updates. Also known as half-precision floating point, FP16 uses 16 bits to represent a number, including its sign, exponent, and fraction. It offers a good balance between precision and memory savings. FP16 is particularly popular in deep learning training on GPUs that support mixed-precision arithmetic, combining the speed benefits of FP16 with the precision of FP32 where needed.

Figure 8.7 shows three different floating-point formats: Float32, Float16, and BFloat16.

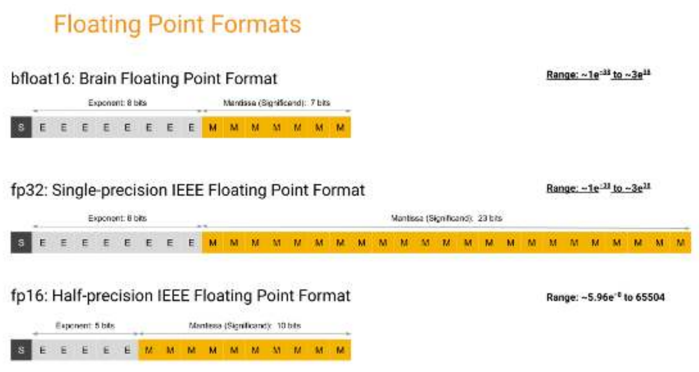

Floating Point Formats
bfloat16: Brain Floating Point Format
Range: ~1e-38 to ~3e38
Exponent: 8 bits
Mantissa (Significand): 7 bits
S E E E E E E E E M M M M M M M
fp32: Single-precision IEEE Floating Point Format
Range: ~1e-38 to ~3e38
Exponent: 8 bits
Mantissa (Significand): 23 bits
S E E E E E E E E M M M M M M M M M M M M M M M M M M M M M M M
fp16: Half-precision IEEE Floating Point Format
Range: ~5.96e-8 to 65504
Exponent: 5 bits
Mantissa (Significand): 10 bits
S E E E E E M M M M M M M M M M

Figure 8.7: Three floating-point formats.

Several other numerical formats fall into an exotic class. An exotic example is BF16 or Brain Floating Point. It is a 16-bit numerical format designed explicitly for deep learning applications. It is a compromise between FP32 and FP16, retaining the 8-bit exponent from FP32 while reducing the mantissa to 7 bits (as compared to FP32's 23-bit mantissa).

This structure prioritizes range over precision. BF16 has achieved training results comparable in accuracy to FP32 while using significantly less memory and computational resources (Kalamkar et al. 2019). This makes it suitable not just for inference but also for training deep neural networks.

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.

Integer: These are integer representations using 8, 4, and 2 bits. They are often used during the inference phase of neural networks, where the weights and activations of the model are quantized to these lower precisions. Integer representations are deterministic and offer significant speed and memory advantages over floating-point representations. For many inference tasks, especially on edge devices, the slight loss in accuracy due to quantization is often acceptable, given the efficiency gains. An extreme form of integer numerics is for binary neural networks (BNNs), where weights and activations are constrained to one of two values: +1 or -1.

Variable bit widths: Beyond the standard widths, research is ongoing into extremely low bit-width numerics, even down to binary or ternary representations. Extremely low bit-width operations can offer significant speedups and further reduce power consumption. While challenges remain in maintaining model accuracy with such drastic quantization, advances continue to be made in this area.

Efficient numerics is not just about reducing the bit-width of numbers but understanding the trade-offs between accuracy and efficiency. As machine learning models become more pervasive, especially in real-world, resource-constrained environments, the focus on efficient numerics will continue to grow. By thoughtfully selecting and leveraging the appropriate numeric precision, one can achieve robust model performance while optimizing for speed, memory, and energy. Table 8.1 summarizes these trade-offs.

Table 8.1: Comparing precision levels in deep learning.

Precision	Pros	Cons
FP32 (Floating Point 32-bit)	• Standard precision used in most deep learning frameworks. • High accuracy due to ample representational capacity. • Well-suited for training	• High memory usage. • Slower inference times compared to quantized models. • Higher energy consumption.
FP16 (Floating Point 16-bit)	• Reduces memory usage compared to FP32. • Speeds up computations on hardware that supports FP16. • Often used in mixed-precision training to balance speed and accuracy.	• Lower representational capacity compared to FP32. • Risk of numerical instability in some models or layers.
INT8 (8-bit Integer)	• Significantly reduced memory footprint compared to floating-point representations. • Faster inference if hardware supports INT8 computations. • Suitable for many post-training quantization scenarios.	• Quantization can lead to some accuracy loss. • Requires careful calibration during quantization to minimize accuracy degradation.
INT4 (4-bit Integer)	• Even lower memory usage than INT8. • Further speedup potential for inference.	• Higher risk of accuracy loss compared to INT8. • Calibration during quantization becomes more critical.

Precision	Pros	Cons
Binary	• Minimal memory footprint (only 1 bit per parameter). • Extremely fast inference due to bitwise operations. • Power efficient.	• Significant accuracy drop for many tasks. • Complex training dynamics due to extreme quantization.
Ternary	• Low memory usage but slightly more than binary. • Offers a middle ground between representation and efficiency.	• Accuracy might still be lower than that of higher precision models. • Training dynamics can be complex.

8.6.2 Efficiency Benefits

Numerical efficiency matters for machine learning workloads for several reasons:

Computational Efficiency : High-precision computations (like FP32 or FP64) can be slow and resource-intensive. Reducing numeric precision can achieve faster computation times, especially on specialized hardware that supports lower precision.

Memory Efficiency: Storage requirements decrease with reduced numeric precision. For instance, FP16 requires half the memory of FP32. This is crucial when deploying models to edge devices with limited memory or working with large models.

Power Efficiency: Lower precision computations often consume less power, which is especially important for battery-operated devices.

Noise Introduction: Interestingly, the noise introduced using lower precision can sometimes act as a regularizer, helping to prevent overfitting in some models.

Hardware Acceleration: Many modern AI accelerators and GPUs are optimized for lower precision operations, leveraging the efficiency benefits of such numerics.

8.7 Evaluating Models

It's worth noting that the actual benefits and trade-offs can vary based on the specific architecture of the neural network, the dataset, the task, and the hardware being used. Before deciding on a numeric precision, it's advisable to perform experiments to evaluate the impact on the desired application.

8.7.1 Efficiency Metrics

A deep understanding of model evaluation methods is important to guide this process systematically. When assessing AI models' effectiveness and suitability for various applications, efficiency metrics come to the forefront.

FLOPs (Floating Point Operations), as introduced in Training, gauge a model's computational demands. For instance, a modern neural network like BERT has billions of FLOPs, which might be manageable on a powerful cloud server but would be taxing on a smartphone. Higher FLOPs can lead to more prolonged inference times and significant power drain, especially on devices without specialized hardware accelerators. Hence, for real-time applications such as video streaming or gaming, models with lower FLOPs might be more desirable.

Memory Usage pertains to how much storage the model requires, affecting both the deploying device's storage and RAM. Consider deploying a model onto a smartphone: a model that occupies several gigabytes of space not only consumes precious storage but might also be slower due to the need to load large weights into memory. This becomes especially crucial for edge devices like security cameras or drones, where minimal memory footprints are vital for storage and rapid data processing.

Power Consumption becomes especially crucial for devices that rely on batteries. For instance, a wearable health monitor using a power-hungry model could drain its battery in hours, rendering it impractical for continuous health monitoring. Optimizing models for low power consumption becomes essential as we move toward an era dominated by IoT devices, where many devices operate on battery power.

Inference Time is about how swiftly a model can produce results. In applications like autonomous driving, where split-second decisions are the difference between safety and calamity, models must operate rapidly. If a self-driving car's model takes even a few seconds too long to recognize an obstacle, the consequences could be dire. Hence, ensuring a model's inference time aligns with the real-time demands of its application is paramount.

In essence, these efficiency metrics are more than numbers dictating where and how a model can be effectively deployed. A model might boast high accuracy, but if its FLOPs, memory usage, power consumption, or inference time make it unsuitable for its intended platform or real-world scenarios, its practical utility becomes limited.

8.7.2 Efficiency Comparisons

The landscape of machine learning models is vast, with each model offering a unique set of strengths and implementation considerations. While raw accuracy figures or training and inference speeds might be tempting benchmarks, they provide an incomplete picture. A deeper comparative analysis reveals several critical factors influencing a model's suitability for TinyML applications. Often, we encounter the delicate balance between accuracy and efficiency. For instance, while a dense, deep learning model and a lightweight MobileNet variant might excel in image classification, their computational demands could be at two extremes. This differentiation is especially pronounced when comparing deployments on resource-abundant cloud servers versus constrained TinyML devices. In many real-world scenarios, the marginal gains in accuracy could be overshadowed by the inefficiencies of a resource-intensive model.

Moreover, the optimal model choice is not always universal but often depends on the specifics of an application. For instance, a model that excels in general object detection scenarios might struggle in niche environments, such as detecting manufacturing defects on a factory floor. This adaptability- or the lack of it- can influence a model's real-world utility.

Another important consideration is the relationship between model complexity and its practical benefits. Take voice-activated assistants, such as "Alexa" or "OK Google." While a complex model might demonstrate a marginally superior understanding of user speech if it's slower to respond than a simpler counterpart, the user experience could be compromised. Thus, adding layers or parameters only sometimes equates to better real-world outcomes.

Another important consideration is the relationship between model complexity and its practical benefits. Take voice-activated assistants like "Alexa" or "OK Google." While a complex model might demonstrate a marginally superior understanding of user speech if it's slower to respond than a simpler counterpart, the user experience could be compromised. Thus, adding layers or parameters only sometimes equates to better real-world outcomes.

Furthermore, while benchmark datasets, such as ImageNet (Russakovsky et al. 2015), COCO (T.-Y. Lin et al. 2014), Visual Wake Words (L. Wang and Zhan 2019a), Google Speech Commands (Warden 2018), etc. provide a standardized performance metric, they might not capture the diversity and unpredictability of real-world data. Two facial recognition models with similar benchmark scores might exhibit varied competencies when faced with diverse ethnic backgrounds or challenging lighting conditions. Such disparities underscore the importance of robustness and consistency across varied data. For example, Figure 8.8 from the Dollar Street dataset shows stove images across extreme monthly incomes. Stoves have different shapes and technological levels across different regions and income levels. A model that is not trained on diverse datasets might perform well on a benchmark but fail in real-world applications. So, if a model was trained on pictures of stoves found in wealthy countries only, it would fail to recognize stoves from poorer regions.

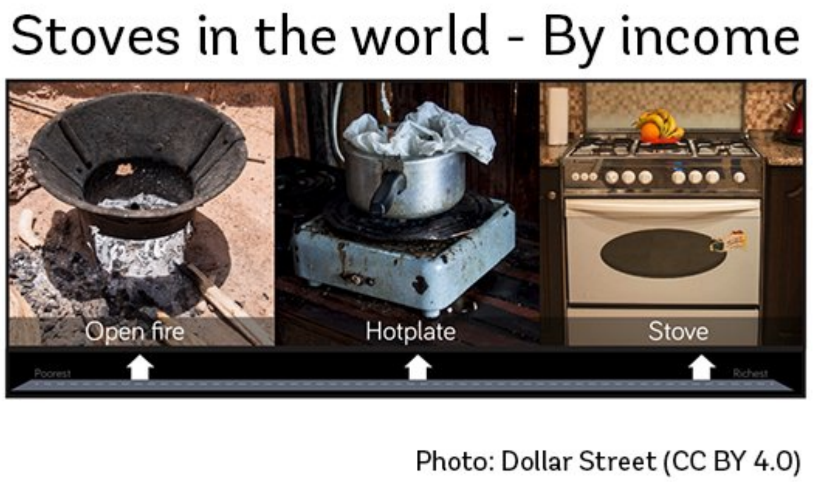

Stoves in the world - By income
Open fire
Hotplate
Stove
Poorest
Richest
Photo: Dollar Street (CC BY 4.0)

Figure 8.8: Different types of stoves. Source: Dollar Street stove images.

In essence, a thorough comparative analysis transcends numerical metrics. It's a holistic assessment intertwined with real-world applications, costs, and the intricate subtleties that each model brings to the table. This is why having standard benchmarks and metrics widely established and adopted by the community becomes important.

8.8 Conclusion

Efficient AI is crucial as we push towards broader and more diverse real-world deployment of machine learning. This chapter provided an overview, exploring the various methodologies and considerations behind achieving efficient AI, starting with the fundamental need, similarities, and differences across cloud, Edge, and TinyML systems.

We examined efficient model architectures and their usefulness for optimization. Model compression techniques such as pruning, quantization, and knowledge distillation exist to help reduce computational demands and memory footprint without significantly impacting accuracy. Specialized hardware like TPUs and NN accelerators offer optimized silicon for neural network operations and data flow. Efficient numerics balance precision and efficiency, enabling models to attain robust performance using minimal resources. We will explore these topics in depth and detail in the subsequent chapters.

Together, these form a holistic framework for efficient AI. But the journey doesn't end here. Achieving optimally efficient intelligence requires continued research and innovation. As models become more sophisticated, datasets grow, and applications diversify into specialized domains, efficiency must evolve in lockstep. Measuring real-world impact requires nuanced benchmarks and standardized metrics beyond simplistic accuracy figures.

Moreover, efficient AI expands beyond technological optimization and encompasses costs, environmental impact, and ethical considerations for the broader societal good. As AI permeates industries and daily lives, a comprehensive outlook on efficiency underpins its sustainable and responsible progress. The subsequent chapters will build upon these foundational concepts, providing actionable insights and hands-on best practices for developing and deploying efficient AI solutions.

8.9 Resources

Here is a curated list of resources to support students and instructors in their learning and teaching journeys. We are continuously working on expanding this collection and will add new exercises soon.

Slides

These slides are a valuable tool for instructors to deliver lectures and for students to review the material at their own pace. We encourage students and instructors to leverage these slides to improve their understanding and facilitate effective knowledge transfer.

- Deploying on Edge Devices: challenges and techniques.
- Model Evaluation.
- Continuous Evaluation Challenges for TinyML.

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.*

Chapter 9

Model Optimizations

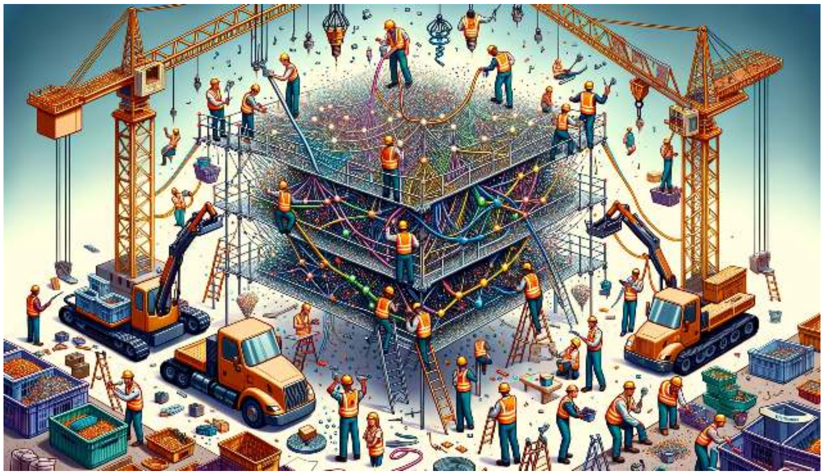


Figure 9.1: *DALL·E 3 Prompt: Illustration of a neural network model represented as a busy construction site, with a diverse group of construction workers, both male and female, of various ethnicities, labeled as 'pruning', 'quantization', and 'sparsity'. They are working together to make the neural 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.*

When machine learning models are deployed on systems, especially on resource-constrained embedded systems, the optimization of models is a necessity. While machine learning inherently often demands substantial computational resources, the systems are inherently limited in memory, processing power, and energy. This chapter will dive into the art and science of optimizing machine learning models to ensure they are lightweight, efficient, and effective when deployed in TinyML scenarios.

Learning Objectives

- Learn techniques like pruning, knowledge distillation and specialized model architectures to represent models more efficiently
- Understand quantization methods to reduce model size and enable faster inference through reduced precision numerics
- Explore hardware-aware optimization approaches to match models to target device capabilities
- Develop holistic thinking to balance tradeoffs in model complexity, accuracy, latency, power etc. based on application requirements
- Discover software tools like frameworks and model conversion platforms that enable deployment of optimized models
- Gain strategic insight into selecting and applying model optimizations based on use case constraints and hardware targets

9.1 Overview

The optimization of machine learning models for practical deployment is a critical aspect of AI systems. This chapter focuses on exploring model optimization techniques as they relate to the development of ML systems, ranging from high-level model architecture considerations to low-level hardware adaptations. Figure 9.2 Illustrates the three layers of the optimization stack we cover.

At the highest level, we examine methodologies for reducing the complexity of model parameters without compromising inferential capabilities. Techniques such as pruning and knowledge distillation offer powerful approaches to compress and refine models while maintaining or even improving their performance, not only in terms of model quality but also in actual system runtime performance. These methods are crucial for creating efficient models that can be deployed in resource-constrained environments.

Furthermore, we explore the role of numerical precision in model computations. Understanding how different levels of numerical preci-