O'REILLY®

Practical Deep Learning for Cloud, Mobile & Edge

Real-World AI & Computer-Vision Projects Using Python, Keras & TensorFlow

Anirudh Koul,
Siddha Ganju & Meher Kasam

Practical Deep Learning for Cloud, Mobile, and Edge

Real-World AI and Computer-Vision Projects Using Python, Keras, and TensorFlow

Anirudh Koul, Siddha Ganju, and Meher Kasam

O'REILLY®

Practical Deep Learning for Cloud, Mobile, and Edge

Real-World AI and Computer-Vision Projects Using Python, Keras, and TensorFlow

Anirudh Koul, Siddha Ganju, and Meher Kasam

O'REILLY®

Practical Deep Learning for Cloud, Mobile, and Edge

by Anirudh Koul, Siddha Ganju, and Meher Kasam

Copyright © 2020 Anirudh Koul, Siddha Ganju, Meher Kasam. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotiona use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rachel Roumeliotis

Development Editor: Nicole Tache

Production Editor: Christopher Faucher

Copyeditor: Octal Publishing, LLC

Proofreader: Christina Edwards

Indexer: Judith McConville

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

October 2019: First Edition

l

Revision History for the First Edition

- 2019-10-14: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492034865* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Practical Deep Learning for Cloud, Mobile, and Edge*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-03486-5

[LSI]

Revision History for the First Edition

- 2019-10-14: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492034865* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Practical Deep Learning for Cloud, Mobile, and Edge*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-03486-5

[LSI]

Preface

We are experiencing a renaissance of artificial intelligence, and everyone and their neighbor wants to be a part of this movement. That's quite likely why you are browsing through this book. There are tons of books about deep learning out there. So you might ask us, very reasonably, why does this book even exist? We'll get to that in just a second.

During our own deep learning journeys since 2013 (while building products at companies including Microsoft, NVIDIA, Amazon, and Square), we witnessed dramatic shifts in this landscape. Constantly evolving research was a given and a lack of mature tooling was a reality of life.

While growing and learning from the community, we noticed a lack of clear guidance on how to convert research to an end product for everyday users. After all, the end user is somewhere in front of a web browser, a smartphone, or an edge device. This often involved countless hours of hacking and experimentation, extensively searching through blogs, GitHub issue threads, and Stack Overflow answers, and emailing authors of packages to get esoteric knowledge, as well as occasional "Aha!" moments. Even the books on the market tended to focus more on theory or how to use a specific tool. The best we could hope to learn from the available books was to build a toy example.

To fill this gap between theory and practice, we originally started giving talks on taking artificial intelligence from research to the end user with a particular focus on practical applications. The talks were structured to showcase motivating examples, as well as different levels of complexity based on skill level (from a hobbyist to a Google-scale engineer) and effort involved in deploying deep learning in production. We discovered that beginners and experts alike found value in these talks.

Over time, the landscape thankfully became accessible to beginners and more tooling became available. Great online material like Fast.ai and DeepLearning.ai made understanding how to train AI models easier than ever. Books also cornered the market on teaching fundamentals using deep learning frameworks such as TensorFlow and PyTorch. But even with all of this, the wide chasm

between theory and production remained largely unaddressed. And we wanted to bridge this gap. Thus, the book you are now reading.

Using approachable language as well as ready-to-run fun projects in computer vision, the book starts off with simple classifiers assuming no knowledge of machine learning and AI, gradually building to add complexity, improve accuracy and speed, scale to millions of users, deploy on a wide variety of hardware and software, and eventually culminate in using reinforcement learning to build a miniature self-driving car.

Nearly every chapter begins with a motivating example, establishes the questions upfront that one might ask through the process of building a solution, and discusses multiple approaches for solving problems, each with varying levels of complexity and effort involved. If you are seeking a quick solution, you might end up just reading a few pages of a chapter and be done. Someone wanting to gain a deeper understanding of the subject should read the entire chapter. Of course, everyone should peruse the case studies included in these chapters for two reasons—they are fun to read and they showcase how people in the industry are using the concepts discussed in the chapter to build real products.

We also discuss many of the practical concerns faced by deep learning practitioners and industry professionals in building real-world applications using the cloud, browsers, mobile, and edge devices. We compiled a number of practical “tips and tricks,” as well as life lessons in this book to encourage our readers to build applications that can make someone’s day just a little bit better.

between theory and production remained largely unaddressed. And we wanted to bridge this gap. Thus, the book you are now reading.

Using approachable language as well as ready-to-run fun projects in computer vision, the book starts off with simple classifiers assuming no knowledge of machine learning and AI, gradually building to add complexity, improve accuracy and speed, scale to millions of users, deploy on a wide variety of hardware and software, and eventually culminate in using reinforcement learning to build a miniature self-driving car.

Nearly every chapter begins with a motivating example, establishes the questions upfront that one might ask through the process of building a solution, and discusses multiple approaches for solving problems, each with varying levels of complexity and effort involved. If you are seeking a quick solution, you might end up just reading a few pages of a chapter and be done. Someone wanting to gain a deeper understanding of the subject should read the entire chapter. Of course, everyone should peruse the case studies included in these chapters for two reasons—they are fun to read and they showcase how people in the industry are using the concepts discussed in the chapter to build real products.

We also discuss many of the practical concerns faced by deep learning practitioners and industry professionals in building real-world applications using the cloud, browsers, mobile, and edge devices. We compiled a number of practical “tips and tricks,” as well as life lessons in this book to encourage our readers to build applications that can make someone’s day just a little bit better.

To the Backend/Frontend/Mobile Software Developer

You are quite likely a proficient programmer already. Even if Python is an unfamiliar language to you, we expect that you will be able to pick it up easily and get started in no time. Best of all, we don't expect you to have any background in machine learning and AI; that's what we are here for! We believe that you will gain value from the book's focus on the following areas:

- How to build user-facing AI products
- How to train models quickly
- How to minimize the code and effort required in prototyping
- How to make models more performant and energy efficient
- How to operationalize and scale, and estimate the costs involved
- Discovering how AI is applied in the industry with 40+ case studies and real-world examples
- Developing a broad-spectrum knowledge of deep learning
- Developing a generalized skill set that can be applied on new frameworks (e.g., PyTorch), domains (e.g., healthcare, robotics), input modalities (e.g., video, audio, text), and tasks (e.g., image segmentation, one-shot learning)

To the Backend/Frontend/Mobile Software Developer

You are quite likely a proficient programmer already. Even if Python is an unfamiliar language to you, we expect that you will be able to pick it up easily and get started in no time. Best of all, we don't expect you to have any background in machine learning and AI; that's what we are here for! We believe that you will gain value from the book's focus on the following areas:

- How to build user-facing AI products
- How to train models quickly
- How to minimize the code and effort required in prototyping
- How to make models more performant and energy efficient
- How to operationalize and scale, and estimate the costs involved
- Discovering how AI is applied in the industry with 40+ case studies and real-world examples
- Developing a broad-spectrum knowledge of deep learning
- Developing a generalized skill set that can be applied on new frameworks (e.g., PyTorch), domains (e.g., healthcare, robotics), input modalities (e.g., video, audio, text), and tasks (e.g., image segmentation, one-shot learning)

To the Data Scientist

You might already be proficient at machine learning and potentially know how to train deep learning models. Good news! You can further enrich your skill set and deepen your knowledge in the field in order to build real products. This book will help inform your everyday work and beyond by covering how to:

- Speed up your training, including on multinode clusters
- Build an intuition for developing and debugging models, including hyperparameter tuning, thus dramatically improving model accuracy
- Understand how your model works, uncover bias in the data, and automatically determine the best hyperparameters as well as model architecture using AutoML
- Learn tips and tricks used by other data scientists, including gathering data quickly, tracking your experiments in an organized manner, sharing your models with the world, and being up to date on the best available models for your task
- Use tools to deploy and scale your best model to real users, even automatically (without involving a DevOps team)

To the Data Scientist

You might already be proficient at machine learning and potentially know how to train deep learning models. Good news! You can further enrich your skill set and deepen your knowledge in the field in order to build real products. This book will help inform your everyday work and beyond by covering how to:

- Speed up your training, including on multinode clusters
- Build an intuition for developing and debugging models, including hyperparameter tuning, thus dramatically improving model accuracy
- Understand how your model works, uncover bias in the data, and automatically determine the best hyperparameters as well as model architecture using AutoML
- Learn tips and tricks used by other data scientists, including gathering data quickly, tracking your experiments in an organized manner, sharing your models with the world, and being up to date on the best available models for your task
- Use tools to deploy and scale your best model to real users, even automatically (without involving a DevOps team)

To the Student

This is a great time to be considering a career in AI—it's turning out to be the next revolution in technology after the internet and smartphones. A lot of strides have been made, and a lot remains to be discovered. We hope that this book can serve as your first step in whetting your appetite for a career in AI and, even better, developing deeper theoretical knowledge. And the best part is that you don't have to spend a lot of money to buy expensive hardware. In fact, you can train on powerful hardware entirely for free from your web browser (thank you, Google Colab!). With this book, we hope you will:

- Aspire to a career in AI by developing a portfolio of interesting projects
- Learn from industry practices to help prepare for internships and job opportunities
- Unleash your creativity by building fun applications like an autonomous ca
- Become an AI for Good champion by using your creativity to solve the most pressing problems faced by humanity

To the Student

This is a great time to be considering a career in AI—it's turning out to be the next revolution in technology after the internet and smartphones. A lot of strides have been made, and a lot remains to be discovered. We hope that this book can serve as your first step in whetting your appetite for a career in AI and, even better, developing deeper theoretical knowledge. And the best part is that you don't have to spend a lot of money to buy expensive hardware. In fact, you can train on powerful hardware entirely for free from your web browser (thank you, Google Colab!). With this book, we hope you will:

- Aspire to a career in AI by developing a portfolio of interesting projects
- Learn from industry practices to help prepare for internships and job opportunities
- Unleash your creativity by building fun applications like an autonomous car
- Become an AI for Good champion by using your creativity to solve the most pressing problems faced by humanity

To the Teacher

We believe that this book can nicely supplement your coursework with fun, real-world projects. We've covered every step of the deep learning pipeline in detail, along with techniques on how to execute each step effectively and efficiently. Each of the projects we present in the book can make for great collaborative or individual work in the classroom throughout the semester. Eventually, we will be releasing PowerPoint Presentation Slides on *http://PracticalDeepLearning.ai* that can accompany coursework.

To the Teacher

We believe that this book can nicely supplement your coursework with fun, real-world projects. We've covered every step of the deep learning pipeline in detail, along with techniques on how to execute each step effectively and efficiently. Each of the projects we present in the book can make for great collaborative or individual work in the classroom throughout the semester. Eventually, we will be releasing PowerPoint Presentation Slides on *http://PracticalDeepLearning.ai* that can accompany coursework.

To the Robotics Enthusiast

Robotics is exciting. If you're a robotics enthusiast, we don't really need to convince you that adding intelligence to robots is the way to go. Increasingly capable hardware platforms such as Raspberry Pi, NVIDIA Jetson Nano, Google Coral, Intel Movidius, PYNQ-Z2, and others are helping drive innovation in the robotics space. As we grow towards Industry 4.0, some of these platforms will become more and more relevant and ubiquitous. With this book, you will:

- Learn how to build and train AI, and then bring it to the edge
- Benchmark and compare edge devices on performance, size, power, battery and costs
- Understand how to choose the optimal AI algorithm and device for a given scenario
- Learn how other makers are building creative robots and machines
- Learn how to build further progress in the field and showcase your work

To the Robotics Enthusiast

Robotics is exciting. If you're a robotics enthusiast, we don't really need to convince you that adding intelligence to robots is the way to go. Increasingly capable hardware platforms such as Raspberry Pi, NVIDIA Jetson Nano, Google Coral, Intel Movidius, PYNQ-Z2, and others are helping drive innovation in the robotics space. As we grow towards Industry 4.0, some of these platforms will become more and more relevant and ubiquitous. With this book, you will:

- Learn how to build and train AI, and then bring it to the edge
- Benchmark and compare edge devices on performance, size, power, battery, and costs
- Understand how to choose the optimal AI algorithm and device for a given scenario
- Learn how other makers are building creative robots and machines
- Learn how to build further progress in the field and showcase your work

What to Expect in Each Chapter

Chapter 1, Exploring the Landscape of Artificial Intelligence

We take a tour of this evolving landscape, from the 1950s to today, analyze the ingredients that make for a perfect deep learning recipe, get familiar with common AI terminology and datasets, and take a peek into the world of responsible AI.

Chapter 2, What's in the Picture: Image Classification with Keras

We delve into the world of image classification in a mere five lines of Keras code. We then learn what neural networks are paying attention to while making predictions by overlaying heatmaps on videos. Bonus: we hear the motivating personal journey of François Chollet, the creator of Keras, illustrating the impact a single individual can have.

Chapter 3, Cats Versus Dogs: Transfer Learning in 30 Lines with Keras

We use transfer learning to reuse a previously trained network on a new custom classification task to get near state-of-the-art accuracy in a matter of minutes. We then slice and dice the results to understand how well it is classifying. Along the way, we build a common machine learning pipeline, which is repurposed throughout the book. Bonus: we hear from Jeremy Howard, cofounder of fast.ai, on how hundreds of thousands of students use transfer learning to jumpstart their AI journey.

Chapter 4, Building a Reverse Image Search Engine: Understanding Embeddings

Like Google Reverse Image Search, we explore how one can use embeddings—a contextual representation of an image to find similar images in under ten lines. And then the fun starts when we explore different strategies and algorithms to speed this up at scale, from thousands to several million images, and making them searchable in microseconds.

Chapter 5, From Novice to Master Predictor: Maximizing Convolutional Neural Network Accuracy

We explore strategies to maximize the accuracy that our classifier can achieve, with the help of a range of tools including TensorBoard, the What-If Tool, tf-explain, TensorFlow Datasets, AutoKeras, and AutoAugment. Along the way, we conduct experiments to develop an intuition of what parameters might or might not work for your AI task.

Chapter 6, Maximizing Speed and Performance of TensorFlow: A Handy Checklist

We take the speed of training and inference into hyperdrive by going through a checklist of 30 tricks to reduce as many inefficiencies as possible and maximize the value of your current hardware.

Chapter 7, Practical Tools, Tips, and Tricks

We diversify our practical skills in a variety of topics and tools, ranging from installation, data collection, experiment management, visualizations, and keeping track of state-of-the-art research all the way to exploring further avenues for building the theoretical foundations of deep learning.

Chapter 8, Cloud APIs for Computer Vision: Up and Running in 15 Minutes

Work smart, not hard. We utilize the power of cloud AI platforms from Google, Microsoft, Amazon, IBM, and Clarifai in under 15 minutes. For tasks not solved with existing APIs, we then use custom classification services to train classifiers without coding. And then we pit them against each other in an open benchmark—you might be surprised who won.

Chapter 9, Scalable Inference Serving on Cloud with TensorFlow Serving and KubeFlow

We take our custom trained model to the cloud/on-premises to scalably serve from hundreds to millions of requests. We explore Flask, Google Cloud ML Engine, TensorFlow Serving, and KubeFlow, showcasing the effort, scenario and cost-benefit analysis.

Chapter 10, AI in the Browser with TensorFlow.js and ml5.js

Every single individual who uses a computer or a smartphone uniformly has access to one software program—their browser. Reach all those users with browser-based deep learning libraries including TensorFlow.js and ml5.js.

Guest author Zaid Alyafeai walks us through techniques and tasks such as body pose estimation, generative adversarial networks (GANs), image-to-image translation with Pix2Pix, and more, running not on a server but in the browser itself. Bonus: hear from key contributors to TensorFlow.js and ml5.js on how the projects incubated.

Chapter 11, Real-Time Object Classification on iOS with Core ML

We explore the landscape of deep learning on mobile, with a sharp focus on the Apple ecosystem with Core ML. We benchmark models on different iPhones, investigate strategies to reduce app size and energy impact, and look into dynamic model deployment, training on device, and how professional apps are built.

Chapter 12, Not Hotdog on iOS with Core ML and Create ML

Silicon Valley's Not Hotdog app (from HBO) is considered the "Hello World" of mobile AI, so we pay tribute by building a real-time version in not one, not two, but three different ways.

Chapter 13, Shazam for Food: Developing Android Apps with TensorFlow Lite and ML Kit

We bring AI to Android with the help of TensorFlow Lite. We then look at cross-platform development using ML Kit (which is built on top of TensorFlow Lite) and Fritz to explore the end-to-end development life cycle for building a self-improving AI app. Along the way we look at model versioning, A/B testing, measuring success, dynamic updates, model optimization, and other topics. Bonus: we get to hear about the rich experience of Pete Warden (technical lead for Mobile and Embedded TensorFlow) in bringing AI to edge devices.

Chapter 14, Building the Purrfect Cat Locator App with TensorFlow Object Detection API

We explore four different methods for locating the position of objects within images. We take a look at the evolution of object detection over the years, and analyze the tradeoffs between speed and accuracy. This builds the base for case studies such as crowd counting, face detection, and autonomous cars.

Chapter 15, Becoming a Maker: Exploring Embedded AI at the Edge

Guest author Sam Sterckval brings deep learning to low-power devices as he showcases a range of AI-capable edge devices with varying processing power and cost including Raspberry Pi, NVIDIA Jetson Nano, Google Coral, Intel Movidius, and PYNQ-Z2 FPGA, opening the doors for robotics and maker projects. Bonus: hear from the NVIDIA Jetson Nano team on how people are building creative robots quickly from their open source recipe book.

Chapter 16, Simulating a Self-Driving Car Using End-to-End Deep Learning with Keras

Using the photorealistic simulation environment of Microsoft AirSim, guest authors Aditya Sharma and Mitchell Spryn guide us in training a virtual car by driving it first within the environment and then teaching an AI model to replicate its behavior. Along the way, this chapter covers a number of concepts that are applicable in the autonomous car industry.

Chapter 17, Building an Autonomous Car in Under an Hour: Reinforcement Learning with AWS DeepRacer

Moving from the virtual to the physical world, guest author Sunil Mallya showcases how AWS DeepRacer, a miniature car, can be assembled, trained, and raced in under an hour. And with the help of reinforcement learning, the car learns to drive on its own, penalizing its mistakes and maximizing success. We learn how to apply this knowledge to races from the Olympics of AI Driving to RoboRace (using full-sized autonomous cars). Bonus: hear from Anima Anandkumar (NVIDIA) and Chris Anderson (founder of DIY Robocars) on where the self-driving automotive industry is headed.

Chapter 15, Becoming a Maker: Exploring Embedded AI at the Edge

Guest author Sam Sterckval brings deep learning to low-power devices as he showcases a range of AI-capable edge devices with varying processing power and cost including Raspberry Pi, NVIDIA Jetson Nano, Google Coral, Intel Movidius, and PYNQ-Z2 FPGA, opening the doors for robotics and maker projects. Bonus: hear from the NVIDIA Jetson Nano team on how people are building creative robots quickly from their open source recipe book.

Chapter 16, Simulating a Self-Driving Car Using End-to-End Deep Learning with Keras

Using the photorealistic simulation environment of Microsoft AirSim, guest authors Aditya Sharma and Mitchell Spryn guide us in training a virtual car by driving it first within the environment and then teaching an AI model to replicate its behavior. Along the way, this chapter covers a number of concepts that are applicable in the autonomous car industry.

Chapter 17, Building an Autonomous Car in Under an Hour: Reinforcement Learning with AWS DeepRacer

Moving from the virtual to the physical world, guest author Sunil Mallya showcases how AWS DeepRacer, a miniature car, can be assembled, trained, and raced in under an hour. And with the help of reinforcement learning, the car learns to drive on its own, penalizing its mistakes and maximizing success. We learn how to apply this knowledge to races from the Olympics of AI Driving to RoboRace (using full-sized autonomous cars). Bonus: hear from Anima Anandkumar (NVIDIA) and Chris Anderson (founder of DIY Robocars) on where the self-driving automotive industry is headed.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://PracticalDeepLearning.ai*. If you have a technical question or a problem using the code examples, please send email to *PracticalDLBook@gmail.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Practical Deep Learning for Cloud, Mobile, and Edge* by Anirudh Koul, Siddha Ganju, and Meher Kasam (O'Reilly). Copyright 2020 Anirudh Koul, Siddha Ganju, Meher Kasam, 978-1-492-03486-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://PracticalDeepLearning.ai*. If you have a technical question or a problem using the code examples, please send email to *PracticalDLBook@gmail.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Practical Deep Learning for Cloud, Mobile, and Edge* by Anirudh Koul, Siddha Ganju, and Meher Kasam (O'Reilly). Copyright 2020 Anirudh Koul, Siddha Ganju, Meher Kasam, 978-1-492-03486-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O’Reilly Online Learning

NOTE

For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

O’Reilly Online Learning

NOTE

For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

O'Reilly has a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/practical-deep-learning*. The authors have a website for this book as well: *http://PracticalDeepLearning.ai*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book; email *PracticalDLBook@gmail.com* to contact the authors about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

O'Reilly has a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/practical-deep-learning*. The authors have a website for this book as well: *http://PracticalDeepLearning.ai*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book; email *PracticalDLBook@gmail.com* to contact the authors about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Acknowledgments

Group Acknowledgments

We'd like to thank the following people for their immense help throughout our journey in writing this book. Without them, this book would not be possible.

This book came to life because of our development editor Nicole Taché's efforts She rooted for us throughout our journey and provided important guidance at each step of the process. She helped us prioritize the right material (believe it or not, the book was going to be even larger!) and ensured that we were on track. She was reader number one for every single draft that we had written, so our goal first and foremost was ensuring that she was able to follow the content, despite her being new to AI. We're immensely grateful for her support.

We also want to thank the rest of the O'Reilly team including our production editor Christopher Faucher who worked tireless hours on a tight schedule to ensure that this book made it to the printing press on time. We are also grateful to our copy editor Bob Russell who really impressed us with his lightning-fast edits and his attention to detail. He made us realize the importance of paying attention to English grammar in school (though a few years too late, we're afraid). We also want to acknowledge Rachel Roumeliotis (VP of Content Strategy) and Olivia MacDonald (Managing Editor for Development) for believing in the project and for offering their continued support.

Huge thanks are in order for our guest authors who brought in their technical expertise to share their passion for this field with our readers. Aditya Sharma and Mitchell Spryn (from Microsoft) showed us that our love for playing video racing games can be put to good use to train autonomous cars by driving them in a simulated environment (with AirSim). Sunil Mallya (from Amazon) helped bring this knowledge to the physical world by demonstrating that all it takes is one hour to assemble and get a miniature autonomous car (AWS DeepRacer) to navigate its way around a track using reinforcement learning. Sam Sterckval (from Edgise) summarized the vast variety of embedded AI hardware available in the market, so we can get a leg up on our next robotics project. And finally, Zaid Alyafeai (from King Fahd University) demonstrated that browsers are no less capable of running serious interactive AI models (with the help of TensorFlow.js and ml5js).

The book is in its current shape because of timely feedback from our amazing technical reviewers, who worked tirelessly on our drafts, pointed out any technical inaccuracies they came across, and gave us suggestions on better conveying our ideas. Due to their feedback (and the ever-changing APIs of TensorFlow), we ended up doing a rewrite of a majority of the book from the original prerelease. We thank Margaret Maynard-Reid (Google Developer Expert for Machine Learning, you might have read her work while reading TensorFlow documentation), Paco Nathan (35+ years industry veteran at Derwin Inc., who introduced Anirudh to the world of public speaking), Andy Petrella (CEO and Founder at Kensu and creator of SparkNotebook whose technical insights stood up to his reputation), and Nikhita Koul (Senior Data Scientist at Adobe who read and suggested improvements after every iteration, effectively reading a few thousand pages, thus making the content significantly more approachable) for their detailed reviews of each chapter. Additionally, we also had a lot of help from reviewers with expertise in specific topics be it AI in the browser, mobile development, or autonomous cars. The chapter-wise reviewer list (in alphabetical order) is as follows:

- Chapter 1: Dharini Chandrasekaran, Sherin Thomas
- Chapter 2: Anuj Sharma, Charles Kozierok, Manoj Parihar, Pankesh Bamotra, Pranav Kant
- Chapter 3: Anuj Sharma, Charles Kozierok, Manoj Parihar, Pankesh Bamotra, Pranav Kant
- Chapter 4: Anuj Sharma, Manoj Parihar, Pankesh Bamotra, Pranav Kant
- Chapter 6: Gabriel Ibagon, Jiri Simsa, Max Katz, Pankesh Bamotra
- Chapter 7: Pankesh Bamotra
- Chapter 8: Deepesh Aggarwal
- Chapter 9: Pankesh Bamotra
- Chapter 10: Brett Burley, Laurent Denoue, Manraj Singh
- Chapter 11: David Apgar, James Webb
- Chapter 12: David Apgar

- Chapter 13: Jesse Wilson, Salman Gadit
- Chapter 14: Akshit Arora, Pranav Kant, Rohit Taneja, Ronay Ak
- Chapter 15: Geertrui Van Lommel, Joke Decubber, Jolien De Cock, Marianne Van Lommel, Sam Hendrickx
- Chapter 16: Dario Salischiker, Kurt Niebuhr, Matthew Chan, Praveen Palanisamy
- Chapter 17: Kirtesh Garg, Larry Pizette, Pierre Dumas, Ricardo Sueiras, Segolene Dessertine-panhard, Sri Elaprolu, Tatsuya Arai

We have short excerpts throughout the book from creators who gave us a little peek into their world, and how and why they built the project for which they are most known. We are grateful to François Chollet, Jeremy Howard, Pete Warden, Anima Anandkumar, Chris Anderson, Shanqing Cai, Daniel Smilkov, Cristobal Valenzuela, Daniel Shiffman, Hart Woolery, Dan Abdinoor, Chitoku Yato, John Welsh, and Danny Atsmo.

- Chapter 13: Jesse Wilson, Salman Gadit
- Chapter 14: Akshit Arora, Pranav Kant, Rohit Taneja, Ronay Ak
- Chapter 15: Geertrui Van Lommel, Joke Decubber, Jolien De Cock, Marianne Van Lommel, Sam Hendrickx
- Chapter 16: Dario Salischiker, Kurt Niebuhr, Matthew Chan, Praveen Palanisamy
- Chapter 17: Kirtesh Garg, Larry Pizette, Pierre Dumas, Ricardo Sueiras, Segolene Dessertine-panhard, Sri Elaprolu, Tatsuya Arai

We have short excerpts throughout the book from creators who gave us a little peek into their world, and how and why they built the project for which they are most known. We are grateful to François Chollet, Jeremy Howard, Pete Warden, Anima Anandkumar, Chris Anderson, Shanqing Cai, Daniel Smilkov, Cristobal Valenzuela, Daniel Shiffman, Hart Woolery, Dan Abdinoor, Chitoku Yato, John Welsh, and Danny Atsmo.

Personal Acknowledgments

"I would like to thank my family—Arbind, Saroj, and Nikhita who gave me the support, resources, time, and freedom to pursue my passions. To all the hackers and researchers at Microsoft, Aira, and Yahoo who stood with me in turning ideas to prototypes to products, it's not the successes but the hiccups which taught us a lot during our journey together. Our trials and tribulations provided glorious material for this book, enough to exceed our original estimate by an extra 250 pages! To my academic families at Carnegie Mellon, Dalhousie, and Thapar University, you taught me more than just academics (unlike what my GPA might suggest). And to the blind and low-vision community, you inspired me daily to work in the AI field by demonstrating that armed with the right tools, people are truly limitless."

—Anirudh

"My grandfather, an author himself, once told me, 'Authoring a book is harder than I thought and more rewarding than I could have ever imagined.' I am eternally grateful to my grandparents and family, Mom, Dad, and Shriya for advocating seeking out knowledge and helping me become the person I am today. To my wonderful collaborators and mentors from Carnegie Mellon University, CERN, NASA FDL, Deep Vision, NITH, and NVIDIA who were with me throughout my journey, I am indebted to them for teaching and helping develop a scientific temperament. To my friends, who I hope still remember me, as I've been pretty invisible as of late, I would like to say a big thanks for being incredibly patient. I hope to see you all around. To my friends who selflessly reviewed chapters of the book and acted as a sounding board, a huge thank you—without you, the book would not have taken shape."

—Siddha

"I am indebted to my parents Rajagopal and Lakshmi for their unending love and support starting from the very beginning and their strong will to provide me with a good life and education. I am grateful to my professors from UF and VNIT who taught me well and made me glad that I majored in CS. I am thankful to my incredibly supportive partner Julia Tanner who, for nearly two years, had to endure nights and weekends of unending Skype calls with my coauthors, as well as my several terrible jokes (some of which unfortunately

made it into this book). I'd also like to acknowledge the role of my wonderful manager Joel Kustka in supporting me during the process of writing this book. A shout out to my friends who were incredibly understanding when I couldn't hang out with them as often as they would have liked me to."

—Meher

Last but not least, thank you to the makers of Grammarly, which empowered people with mediocre English grades to become published authors!

made it into this book). I'd also like to acknowledge the role of my wonderful manager Joel Kustka in supporting me during the process of writing this book. A shout out to my friends who were incredibly understanding when I couldn't hang out with them as often as they would have liked me to."

—Meher

Last but not least, thank you to the makers of Grammarly, which empowered people with mediocre English grades to become published authors!

Chapter 1. Exploring the Landscape of Artificial Intelligence

Following are the words from Dr. May Carson's (Figure 1-1) seminal paper on the changing role of artificial intelligence (AI) in human life in the twenty-first century:

> *Artificial Intelligence has often been termed as the electricity of the 21st century. Today, artificial intelligent programs will have the power to drive all forms of industry (including health), design medical devices and build new types of products and services, including robots and automobiles. As AI is advancing, organizations are already working to ensure those artificial intelligence programs can do their job and, importantly, avoid mistakes or dangerous accidents. Organizations need AI, but they also recognize that not everything they can do with AI is a good idea.*
>
> *We have had extensive studies of what it takes to operate artificial intelligence using these techniques and policies. The main conclusion is that the amount of money spent on AI programs per person, per year versus the amount used to research, build and produce them is roughly equal. That seems obvious, but it's not entirely true.*
>
> *First, AI systems need support and maintenance to help with their functions. In order to be truly reliable, they need people to have the skills to run them and to help them perform some of their tasks. It's essential that AI organizations provide workers to do the complex tasks needed by those services. It's also important to understand the people who are doing those jobs, especially once AI is more complex than humans. For example, people will most often work in jobs requiring advanced knowledge but are not necessarily skilled in working with systems that need to be built and maintained.*

Figure 1-1. Dr. May Carson

Figure 1-1. Dr. May Carson

An Apology

We now have to come clean and admit that everything in this chapter up to now was entirely fake. Literally everything! All of the text (other than the first italicized sentence, which was written by us as a seed) was generated using the GPT-2 model (built by Adam King) on the website *TalkToTransformer.com*. The name of the author was generated using the "Nado Name Generator" on the website *Onitools.moe*. At least the picture of the author must be real, right? Nope, the picture was generated from the website *ThisPersonDoesNotExist.com* which shows us new pictures of nonexistent people each time we reload the page using the magic of Generative Adversarial Networks (GANs).

Although we feel ambivalent, to say the least, about starting this entire book on a dishonest note, we thought it was important to showcase the state-of-the-art of AI when you, our reader, least expected it. It is, frankly, mind-boggling and amazing and terrifying at the same time to see what AI is already capable of. The fact that it can create sentences out of thin air that are more intelligent and eloquent than some world leaders is...let's just say big league.

That being said, one thing AI can't appropriate from us just yet is the ability to be fun. We're hoping that those first three fake paragraphs will be the driest in this entire book. After all, we don't want to be known as "the authors more boring than a machine."

An Apology

We now have to come clean and admit that everything in this chapter up to now was entirely fake. Literally everything! All of the text (other than the first italicized sentence, which was written by us as a seed) was generated using the GPT-2 model (built by Adam King) on the website *TalkToTransformer.com*. The name of the author was generated using the "Nado Name Generator" on the website *Onitools.moe*. At least the picture of the author must be real, right? Nope, the picture was generated from the website *ThisPersonDoesNotExist.com* which shows us new pictures of nonexistent people each time we reload the page using the magic of Generative Adversarial Networks (GANs).

Although we feel ambivalent, to say the least, about starting this entire book on a dishonest note, we thought it was important to showcase the state-of-the-art of AI when you, our reader, least expected it. It is, frankly, mind-boggling and amazing and terrifying at the same time to see what AI is already capable of. The fact that it can create sentences out of thin air that are more intelligent and eloquent than some world leaders is...let's just say big league.

That being said, one thing AI can't appropriate from us just yet is the ability to be fun. We're hoping that those first three fake paragraphs will be the driest in this entire book. After all, we don't want to be known as "the authors more boring than a machine."

The Real Introduction

Recall that time you saw a magic show during which a trick dazzled you enough to think, “How the heck did they do that?!” Have you ever wondered the same about an AI application that made the news? In this book, we want to equip you with the knowledge and tools to not only deconstruct but also build a similar one.

Through accessible, step-by-step explanations, we dissect real-world applications that use AI and showcase how you would go about creating them on a wide variety of platforms—from the cloud to the browser to smartphones to edge AI devices, and finally landing on the ultimate challenge currently in AI: autonomous cars.

In most chapters, we begin with a motivating problem and then build an end-to-end solution one step at a time. In the earlier portions of the book, we develop the necessary skills to build the brains of the AI. But that’s only half the battle. The true value of building AI is in creating usable applications. And we’re not talking about toy prototypes here. We want you to construct software that can be used in the real world by real people for improving their lives. Hence, the word “Practical” in the book title. To that effect, we discuss various options that are available to us and choose the appropriate options based on performance, energy consumption, scalability, reliability, and privacy trade-offs.

In this first chapter, we take a step back to appreciate this moment in AI history. We explore the meaning of AI, specifically in the context of deep learning and the sequence of events that led to deep learning becoming one of the most groundbreaking areas of technological progress in the early twenty-first century. We also examine the core components underlying a complete deep learning solution, to set us up for the subsequent chapters in which we actually get our hands dirty.

So our journey begins here, with a very fundamental question.

The Real Introduction

Recall that time you saw a magic show during which a trick dazzled you enough to think, “How the heck did they do that?!” Have you ever wondered the same about an AI application that made the news? In this book, we want to equip you with the knowledge and tools to not only deconstruct but also build a similar one.

Through accessible, step-by-step explanations, we dissect real-world applications that use AI and showcase how you would go about creating them on a wide variety of platforms—from the cloud to the browser to smartphones to edge AI devices, and finally landing on the ultimate challenge currently in AI: autonomous cars.

In most chapters, we begin with a motivating problem and then build an end-to-end solution one step at a time. In the earlier portions of the book, we develop the necessary skills to build the brains of the AI. But that’s only half the battle. The true value of building AI is in creating usable applications. And we’re not talking about toy prototypes here. We want you to construct software that can be used in the real world by real people for improving their lives. Hence, the word “Practical” in the book title. To that effect, we discuss various options that are available to us and choose the appropriate options based on performance, energy consumption, scalability, reliability, and privacy trade-offs.

In this first chapter, we take a step back to appreciate this moment in AI history. We explore the meaning of AI, specifically in the context of deep learning and the sequence of events that led to deep learning becoming one of the most groundbreaking areas of technological progress in the early twenty-first century. We also examine the core components underlying a complete deep learning solution, to set us up for the subsequent chapters in which we actually get our hands dirty.

So our journey begins here, with a very fundamental question.

What Is AI?

Throughout this book, we use the terms “artificial intelligence,” “machine learning,” and “deep learning” frequently, sometimes interchangeably. But in the strictest technical terms, they mean different things. Here’s a synopsis of each (see also Figure 1-2):

AI

This gives machines the capabilities to mimic human behavior. IBM’s Deep Blue is a recognizable example of AI.

Machine learning

This is the branch of AI in which machines use statistical techniques to learn from previous information and experiences. The goal is for the machine to take action in the future based on learning observations from the past. If you watched IBM’s Watson take on Ken Jennings and Brad Rutter on *Jeopardy!*, you saw machine learning in action. More relatably, the next time a spam email doesn’t reach your inbox, you can thank machine learning.

Deep learning

This is a subfield of machine learning in which deep, multilayered neural networks are used to make predictions, especially excelling in computer vision, speech recognition, natural language understanding, and so on.

Figure 1-2. The relationship between AI, machine learning, and deep learning

Throughout this book, we primarily focus on deep learning.

Figure 1-2. The relationship between AI, machine learning, and deep learning

Throughout this book, we primarily focus on deep learning.

Motivating Examples

Let's cut to the chase. What compelled us to write this book? Why did you spend your hard-earned money[1] buying this book? Our motivation was simple: to get more people involved in the world of AI. The fact that you're reading this book means that our job is already halfway done.

However, to really pique your interest, let's take a look at some stellar examples that demonstrate what AI is already capable of doing:

- "DeepMind's AI agents conquer human pros at StarCraft II": *The Verge*, 2019
- "AI-Generated Art Sells for Nearly Half a Million Dollars at Christie's": *AdWeek*, 2018
- "AI Beats Radiologists in Detecting Lung Cancer": *American Journal of Managed Care*, 2019
- "Boston Dynamics Atlas Robot Can Do Parkour": *ExtremeTech*, 2018
- "Facebook, Carnegie Mellon build first AI that beats pros in 6-player poker": *ai.facebook.com*, 2019
- "Blind users can now explore photos by touch with Microsoft's Seeing AI": *TechCrunch*, 2019
- "IBM's Watson supercomputer defeats humans in final Jeopardy match": *VentureBeat*, 2011
- "Google's ML-Jam challenges musicians to improvise and collaborate with AI": *VentureBeat*, 2019
- "Mastering the Game of Go without Human Knowledge": *Nature*, 2017
- "Chinese AI Beats Doctors in Diagnosing Brain Tumors": *Popular Mechanics*, 2018
- "Two new planets discovered using artificial intelligence": *Phys.org*, 2019
- "Nvidia's latest AI software turns rough doodles into realistic landscapes":

The Verge, 2019

These applications of AI serve as our North Star. The level of these achievements is the equivalent of a gold-medal-winning Olympic performance. However, applications solving a host of problems in the real world is the equivalent of completing a 5K race. Developing these applications doesn't require years of training, yet doing so provides the developer immense satisfaction when crossing the finish line. We are here to coach you through that 5K.

Throughout this book, we intentionally prioritize breadth. The field of AI is changing so quickly that we can only hope to equip you with the proper mindset and array of tools. In addition to tackling individual problems, we will look at how different, seemingly unrelated problems have fundamental overlaps that we can use to our advantage. As an example, sound recognition uses Convolutional Neural Networks (CNNs), which are also the basis for modern computer vision. We touch upon practical aspects of multiple areas so you will be able to go from 0 to 80 quickly to tackle real-world problems. If we've generated enough interes that you decide you then want to go from 80 to 95, we'd consider our goal achieved. As the oft-used phrase goes, we want to "democratize AI."

It's important to note that much of the progress in AI happened in just the past few years—it's difficult to overstate that. To illustrate how far we've come along, take this example: five years ago, you needed a Ph.D. just to get your foot in the door of the industry. Five years later, you don't even need a Ph.D. to write an entire book on the subject. (Seriously, check our profiles!)

Although modern applications of deep learning seem pretty amazing, they did not get there all on their own. They stood on the shoulders of many giants of the industry who have been pushing the limits for decades. Indeed, we can't fully appreciate the significance of this time without looking at the entire history.

The Verge, 2019

These applications of AI serve as our North Star. The level of these achievements is the equivalent of a gold-medal-winning Olympic performance. However, applications solving a host of problems in the real world is the equivalent of completing a 5K race. Developing these applications doesn't require years of training, yet doing so provides the developer immense satisfaction when crossing the finish line. We are here to coach you through that 5K.

Throughout this book, we intentionally prioritize breadth. The field of AI is changing so quickly that we can only hope to equip you with the proper mindset and array of tools. In addition to tackling individual problems, we will look at how different, seemingly unrelated problems have fundamental overlaps that we can use to our advantage. As an example, sound recognition uses Convolutional Neural Networks (CNNs), which are also the basis for modern computer vision. We touch upon practical aspects of multiple areas so you will be able to go from 0 to 80 quickly to tackle real-world problems. If we've generated enough interest that you decide you then want to go from 80 to 95, we'd consider our goal achieved. As the oft-used phrase goes, we want to "democratize AI."

It's important to note that much of the progress in AI happened in just the past few years—it's difficult to overstate that. To illustrate how far we've come along, take this example: five years ago, you needed a Ph.D. just to get your foot in the door of the industry. Five years later, you don't even need a Ph.D. to write an entire book on the subject. (Seriously, check our profiles!)

Although modern applications of deep learning seem pretty amazing, they did not get there all on their own. They stood on the shoulders of many giants of the industry who have been pushing the limits for decades. Indeed, we can't fully appreciate the significance of this time without looking at the entire history.

A Brief History of AI

Let’s go back in time a little bit: our whole universe was in a hot dense state. Then nearly 14 billion years ago expansion started, wait...okay, we don’t have to go back that far (but now we have the song stuck in your head for the rest of the day, right?). It was really just 70 years ago when the first seeds of AI were planted. Alan Turing, in his 1950 paper, “Computing Machinery and Intelligence,” first asked the question “Can machines think?” This really gets into a larger philosophical debate of sentience and what it means to be human. Does it mean to possess the ability to compose a concerto and know that you’ve composed it? Turing found that framework rather restrictive and instead proposed a test: if a human cannot distinguish a machine from another human, does it really matter? An AI that can mimic a human is, in essence, human.

A Brief History of AI

Let’s go back in time a little bit: our whole universe was in a hot dense state. Then nearly 14 billion years ago expansion started, wait...okay, we don’t have to go back that far (but now we have the song stuck in your head for the rest of the day, right?). It was really just 70 years ago when the first seeds of AI were planted. Alan Turing, in his 1950 paper, “Computing Machinery and Intelligence,” first asked the question “Can machines think?” This really gets into a larger philosophical debate of sentience and what it means to be human. Does it mean to possess the ability to compose a concerto and know that you’ve composed it? Turing found that framework rather restrictive and instead proposed a test: if a human cannot distinguish a machine from another human, does it really matter? An AI that can mimic a human is, in essence, human.

Exciting Beginnings

The term “artificial intelligence” was coined by John McCarthy in 1956 at the Dartmouth Summer Research Project. Physical computers weren’t even really a thing back then, so it’s remarkable that they were able to discuss futuristic areas such as language simulation, self-improving learning machines, abstractions on sensory data, and more. Much of it was theoretical, of course. This was the first time that AI became a field of research rather than a single project.

The paper “Perceptron: A Perceiving and Recognizing Automaton” in 1957 by Frank Rosenblatt laid the foundation for deep neural networks. He postulated that it should be feasible to construct an electronic or electromechanical system that will learn to recognize similarities between patterns of optical, electrical, or tonal information. This system would function similar to the human brain. Rather than using a rule-based model (which was standard for the algorithms at the time), he proposed using statistical models to make predictions.

NOTE

Throughout this book, we repeat the phrase *neural network*. What is a neural network? It is a simplified model of the human brain. Much like the brain, it has *neurons* that activate when encountering something familiar. The different neurons are connected via connections (corresponding to synapses in our brain) that help information flow from one neuron to another.

In Figure 1-3, we can see an example of the simplest neural network: a perceptron. Mathematically, the perceptron can be expressed as follows:

$output = f(x_1, x_2, x_3) = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$

Figure 1-3. An example of a perceptron

In 1965, Ivakhnenko and Lapa published the first working neural network in their paper "Group Method of Data Handling—A Rival Method of Stochastic Approximation." There is some controversy in this area, but Ivakhnenko is regarded by some as the father of deep learning.

Around this time, bold predictions were made about what machines would be capable of doing. Machine translation, speech recognition, and more would be performed better than humans. Governments around the world were excited and began opening up their wallets to fund these projects. This gold rush started in the late 1950s and was alive and well into the mid-1970s.

Figure 1-3. An example of a perceptron

In 1965, Ivakhnenko and Lapa published the first working neural network in their paper “Group Method of Data Handling—A Rival Method of Stochastic Approximation.” There is some controversy in this area, but Ivakhnenko is regarded by some as the father of deep learning.

Around this time, bold predictions were made about what machines would be capable of doing. Machine translation, speech recognition, and more would be performed better than humans. Governments around the world were excited and began opening up their wallets to fund these projects. This gold rush started in the late 1950s and was alive and well into the mid-1970s.

The Cold and Dark Days

With millions of dollars invested, the first systems were put into practice. It turned out that a lot of the original prophecies were unrealistic. Speech recognition worked only if it was spoken in a certain way, and even then, only for a limited set of words. Language translation turned out to be heavily erroneous and much more expensive than what it would cost a human to do. Perceptrons (essentially single-layer neural networks) quickly hit a cap for making reliable predictions. This limited their usefulness for most problems in the real world. This is because they are linear functions, whereas problems in the real world often require a nonlinear classifier for accurate predictions. Imagine trying to fit a line to a curve!

So what happens when you over-promise and under-deliver? You lose funding. The Defense Advanced Research Project Agency, commonly known as DARPA (yeah, those people; the ones who built the ARPANET, which later became the internet), funded a lot of the original projects in the United States. However, the lack of results over nearly two decades increasingly frustrated the agency. It was easier to land a man on the moon than to get a usable speech recognizer!

Similarly, across the pond, the Lighthill Report was published in 1974, which said, “The general-purpose robot is a mirage.” Imagine being a Brit in 1974 watching the bigwigs in computer science debating on the BBC as to whether AI is a waste of resources. As a consequence, AI research was decimated in the United Kingdom and subsequently across the world, destroying many careers in the process. This phase of lost faith in AI lasted about two decades and came to be known as the “AI Winter.” If only Ned Stark had been around back then to warn them.

The Cold and Dark Days

With millions of dollars invested, the first systems were put into practice. It turned out that a lot of the original prophecies were unrealistic. Speech recognition worked only if it was spoken in a certain way, and even then, only for a limited set of words. Language translation turned out to be heavily erroneous and much more expensive than what it would cost a human to do. Perceptrons (essentially single-layer neural networks) quickly hit a cap for making reliable predictions. This limited their usefulness for most problems in the real world. This is because they are linear functions, whereas problems in the real world often require a nonlinear classifier for accurate predictions. Imagine trying to fit a line to a curve!

So what happens when you over-promise and under-deliver? You lose funding. The Defense Advanced Research Project Agency, commonly known as DARPA (yeah, those people; the ones who built the ARPANET, which later became the internet), funded a lot of the original projects in the United States. However, the lack of results over nearly two decades increasingly frustrated the agency. It was easier to land a man on the moon than to get a usable speech recognizer!

Similarly, across the pond, the Lighthill Report was published in 1974, which said, “The general-purpose robot is a mirage.” Imagine being a Brit in 1974 watching the bigwigs in computer science debating on the BBC as to whether AI is a waste of resources. As a consequence, AI research was decimated in the United Kingdom and subsequently across the world, destroying many careers in the process. This phase of lost faith in AI lasted about two decades and came to be known as the “AI Winter.” If only Ned Stark had been around back then to warn them.

A Glimmer of Hope

Even during those freezing days, there was some groundbreaking work done in this field. Sure, perceptrons—being linear functions—had limited capabilities. How could one fix that? By chaining them in a network, such that the output of one (or more) perceptron is the input to one (or more) perceptron. In other words, a multilayer neural network, as illustrated in Figure 1-4. The higher the number of layers, the more the nonlinearity it would learn, resulting in better predictions. There is just one issue: how does one train it? Enter Geoffrey Hinton and friends. They published a technique called *backpropagation* in 1986 in the paper "Learning representations by back-propagating errors." How does it work? Make a prediction, see how far off the prediction is from reality, and propagate back the magnitude of the error into the network so it can learn to fix it. You repeat this process until the error becomes insignificant. A simple yet powerful concept. We use the term backpropagation repeatedly throughout this book.

Figure 1-4. An example multilayer neural network (image source)

In 1989, George Cybenko provided the first proof of the *Universal Approximation Theorem*, which states that a neural network with a single hidden layer is theoretically capable of modeling any problem. This was remarkable because it meant that neural networks could (at least in theory) outdo any machine learning approach. Heck, it could even mimic the human brain. But all of this was only on paper. The size of this network would quickly pose limitations in the real world. This could be overcome somewhat by using multiple hidden layers and training the network with…wait for it… backpropagation!

On the more practical side of things, a team at Carnegie Mellon University built the first-ever autonomous vehicle, NavLab 1, in 1986 (Figure 1-5). It initially used a single-layer neural network to control the angle of the steering wheel. This eventually led to NavLab 5 in 1995. During a demonstration, a car drove al but 50 of the 2,850-mile journey from Pittsburgh to San Diego on its own. NavLab got its driver's license before many Tesla engineers were even born!

Figure 1-5. The autonomous NavLab 1 from 1986 in all its glory (image source)

Another standout example from the 1980s was at the United States Postal Service (USPS). The service needed to sort postal mail automatically according to the postal codes (ZIP codes) they were addressed to. Because a lot of the mail has always been handwritten, optical character recognition (OCR) could not be used. To solve this problem, Yann LeCun et al. used handwritten data from the National Institute of Standards and Technology (NIST) to demonstrate that neural networks were capable of recognizing these handwritten digits in their paper "Backpropagation Applied to Handwritten Zip Code Recognition." The agency's network, LeNet, became what the USPS used for decades to automatically scan and sort the mail. This was remarkable because it was the first convolutional neural network that really worked in the wild. Eventually, in the 1990s, banks would use an evolved version of the model called LeNet-5 to read handwritten numbers on checks. This laid the foundation for modern computer vision.

NOTE

Those of you who have read about the MNIST dataset might have already noticed a connection to the

NIST mention we just made. That is because the MNIST dataset essentially consists of a subset of images from the original NIST dataset that had some modifications (the “M” in “MNIST”) applied to them to ease the train and test process for the neural network. Modifications, some of which you can see in Figure 1-6, included resizing them to 28 x 28 pixels, centering the digit in that area, antialiasing, and so on.

Figure 1-6. A sample of handwritten digits from the MNIST dataset

A few others kept their research going, including Jürgen Schmidhuber, who proposed networks like the Long Short-Term Memory (LSTM) with promising applications for text and speech.

At that point, even though the theories were becoming sufficiently advanced, results could not be demonstrated in practice. The main reason was that it was too computationally expensive for the hardware back then and scaling them for larger tasks was a challenge. Even if by some miracle the hardware was available, the data to realize its full potential was certainly not easy to come by. After all, the internet was still in its dial-up phase. Support Vector Machines (SVMs), a machine learning technique introduced for classification problems in 1995, were faster and provided reasonably good results on smaller amounts of data, and thus had become the norm.

As a result, AI and deep learning’s reputation was poor. Graduate students were warned against doing deep learning research because this is the field “where smart scientists would see their careers end.” People and companies working in the field would use alternative words like informatics, cognitive systems,

intelligent agents, machine learning, and others to dissociate themselves from the AI name. It’s a bit like when the U.S. Department of War was rebranded as the Department of Defense to be more palatable to the people.

intelligent agents, machine learning, and others to dissociate themselves from the AI name. It’s a bit like when the U.S. Department of War was rebranded as the Department of Defense to be more palatable to the people.

How Deep Learning Became a Thing

Luckily for us, the 2000s brought high-speed internet, smartphone cameras, video games, and photo-sharing sites like Flickr and Creative Commons (bringing the ability to legally reuse other people's photos). People in massive numbers were able to quickly take photos with a device in their pockets and ther instantly upload. The data lake was filling up, and gradually there were ample opportunities to take a dip. The 14 million-image ImageNet dataset was born from this happy confluence and some tremendous work by (then Princeton's) Fei-Fei Li and company.

During the same decade, PC and console gaming became really serious. Gamers demanded better and better graphics from their video games. This, in turn, pushed Graphics Processing Unit (GPU) manufacturers such as NVIDIA to keep improving their hardware. The key thing to remember here is that GPUs are damn good at matrix operations. Why is that the case? Because the math demands it! In computer graphics, common tasks such as moving objects, rotating them, changing their shape, adjusting their lighting, and so on all use matrix operations. And GPUs specialize in doing them. And you know what else needs a lot of matrix calculations? Neural networks. It's one big happy coincidence.

With ImageNet ready, the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was set up in 2010 to openly challenge researchers to come up with better techniques for classifying this data. A subset of 1,000 categories consisting of approximately 1.2 million images was available to push the boundaries of research. The state-of-the-art computer-vision techniques like Scale-Invariant Feature Transform (SIFT) + SVM yielded a 28% (in 2010) and a 25% (2011) top-5 error rate (i.e., if one of the top five guesses ranked by probability matches, it's considered accurate).

And then came 2012, with an entry on the leaderboard that nearly halved the error rate down to 16%. Alex Krizhevsky, Ilya Sutskever (who eventually founded OpenAI), and Geoffrey Hinton from the University of Toronto submitted that entry. Aptly called AlexNet, it was a CNN that was inspired by LeNet-5. Even at just eight layers, AlexNet had a massive 60 million parameters and 650,000 neurons, resulting in a 240 MB model. It was trained over one week

using two NVIDIA GPUs. This single event took everyone by surprise, proving the potential of CNNs that snowballed into the modern deep learning era.

Figure 1-7 quantifies the progress that CNNs have made in the past decade. We saw a 40% year-on-year decrease in classification error rate among ImageNet LSVRC–winning entries since the arrival of deep learning in 2012. As CNNs grew deeper, the error continued to decrease.

Figure 1-7. Evolution of winning entries at ImageNet LSVRC

Keep in mind we are vastly simplifying the history of AI, and we are surely glossing over some of the details. Essentially, it was a confluence of data, GPUs, and better techniques that led to this modern era of deep learning. And the progress kept expanding further into newer territories. As Table 1-1 highlights, what was in the realm of science fiction is already a reality.

Table 1-1. A highlight reel of the modern deep learning era

2012	Neural network from Google Brain team starts recognizing cats after watching YouTube videos
2013	• Researchers begin tinkering with deep learning on a variety of tasks • word2vec brings context to words and phrases, getting one step closer to understanding meanings • Error rate for speech recognition went down 25%
2014	• GANs invented • Skype translates speech in real time

	• Eugene Goostman, a chatbot, passes the Turing Test • Sequence-to-sequence learning with neural networks invented • Image captioning translates images to sentences
2015	• Microsoft ResNet beats humans in image accuracy, trains 1,000-layer network • Baidu’s Deep Speech 2 does end-to-end speech recognition • Gmail launches Smart Reply • YOLO (You Only Look Once) does object detection in real time • Visual Question Answering allows asking questions based on images
2016	• AlphaGo wins against professional human Go players • Google WaveNets help generate realistic audio • Microsoft achieves human parity in conversational speech recognition
2017	• AlphaGo Zero learns to play Go itself in 3 days • Capsule Nets fix flaws in CNNs • Tensor Processing Units (TPUs) introduced • California allows sale of autonomous cars • Pix2Pix allows generating images from sketches
2018	• AI designs AI better than humans with Neural Architecture Search • Google Duplex demo makes restaurant reservations on our behalf • Deepfakes swap one face for another in videos • Google’s BERT succeeds humans in language understanding tasks • DawnBench and MLPerf established to benchmark AI training
2019	• OpenAI Five crushes Dota2 world champions • StyleGan generates photorealistic images • OpenAI GPT-2 generates realistic text passages • Fujitsu trains ImageNet in 75 seconds • Microsoft invests $1 billion in OpenAI • AI by the Allen Institute passes 12th-grade science test with 80% score

Hopefully, you now have a historical context of AI and deep learning and have an understanding of why this moment in time is significant. It's important to recognize the rapid rate at which progress is happening in this area. But as we have seen so far, this was not always the case.

The original estimate for achieving real-world computer vision was "one summer" back in the 1960s, according to two of the field's pioneers. They were off by only half a century! It's not easy being a futurist. A study by Alexander Wissner-Gross observed that it took 18 years on average between when an algorithm was proposed and the time it led to a breakthrough. On the other hand, that gap was a mere three years on average between when a dataset was made available and the breakthrough it helped achieve! Look at any of the breakthroughs in the past decade. The dataset that enabled that breakthrough was very likely made available just a few years prior.

Data was clearly the limiting factor. This shows the crucial role that a good dataset can play for deep learning. However, data is not the only factor. Let's look at the other pillars that make up the foundation of the perfect deep learning solution.

Hopefully, you now have a historical context of AI and deep learning and have an understanding of why this moment in time is significant. It's important to recognize the rapid rate at which progress is happening in this area. But as we have seen so far, this was not always the case.

The original estimate for achieving real-world computer vision was "one summer" back in the 1960s, according to two of the field's pioneers. They were off by only half a century! It's not easy being a futurist. A study by Alexander Wissner-Gross observed that it took 18 years on average between when an algorithm was proposed and the time it led to a breakthrough. On the other hand, that gap was a mere three years on average between when a dataset was made available and the breakthrough it helped achieve! Look at any of the breakthroughs in the past decade. The dataset that enabled that breakthrough was very likely made available just a few years prior.

Data was clearly the limiting factor. This shows the crucial role that a good dataset can play for deep learning. However, data is not the only factor. Let's look at the other pillars that make up the foundation of the perfect deep learning solution.

Recipe for the Perfect Deep Learning Solution

Before Gordon Ramsay starts cooking, he ensures he has all of the ingredients ready to go. The same goes for solving a problem using deep learning (Figure 1-8).

Figure 1-8. Ingredients for the perfect deep learning solution

And here's your deep learning *mise en place*!

```
Dataset + Model + Framework + Hardware = Deep Learning Solution
```

Let's look into each of these in a little more detail.

Datasets

Just like Pac-Man is hungry for dots, deep learning is hungry for data—lots and lots of data. It needs this amount of data to spot meaningful patterns that can help make robust predictions. Traditional machine learning was the norm in the 1980s and 1990s because it would function even with few hundreds to thousands of examples. In contrast, Deep Neural Networks (DNNs), when built from scratch, would need orders more data for typical prediction tasks. The upside here is far better predictions.

In this century, we are having a data explosion with quintillions of bytes of data being created every single day—images, text, videos, sensor data, and more. But to make effective use of this data, we need labels. To build a sentiment classifier to know whether an Amazon review is positive or negative, we need thousands of sentences and an assigned emotion for each. To train a face segmentation system for a Snapchat lens, we need the precise location of eyes, lips, nose, and so forth on thousands of images. To train a self-driving car, we need video segments labeled with the human driver's reactions on controls such as the brakes, accelerator, steering wheel, and so forth. These labels act as teachers to our AI and are far more valuable than unlabeled data alone.

Getting labels can be pricey. It's no wonder that there is an entire industry around crowdsourcing labeling tasks among thousands of workers. Each label might cost from a few cents to dollars, depending on the time spent by the workers to assign it. For example, during the development of the Microsoft COCO (Common Objects in Context) dataset, it took roughly three seconds to label the name of each object in an image, approximately 30 seconds to place a bounding box around each object, and 79 seconds to draw the outlines for each object. Repeat that hundreds of thousands of times and you can begin to fathom the costs around some of the larger datasets. Some labeling companies like Appen and Scale AI are already valued at more than a billion dollars each.

We might not have a million dollars in our bank account. But luckily for us, two good things happened in this deep learning revolution:

- Gigantic labeled datasets have been generously made public by major companies and universities.

- A technique called *transfer learning*, which allows us to tune our models to datasets with even hundreds of examples—as long as our model was originally trained on a larger dataset similar to our current set. We use this repeatedly in the book, including in Chapter 5 where we experiment and prove even a few tens of examples can get us decent performance with this technique. Transfer learning busts the myth that big data is necessary for training a good model. Welcome to the world of *tiny data*!

Table 1-2 showcases some of the popular datasets out there today for a variety of deep learning tasks.

Table 1-2. A diverse range of public datasets

Data type	Name	Details
Image	Open Images V4 (from Google)	• Nine million images in 19,700 categories • 1.74 Million images with 600 categories (bounding boxes)
	Microsoft COCO	• 330,000 images with 80 object categories • Contains bounding boxes, segmentation, and five captions per image
Video	YouTube-8M	• 6.1 million videos, 3,862 classes, 2.6 billion audio-visual features • 3.0 labels/video • 1.53 TB of randomly sampled videos
Video, images	BDD100K (from UC Berkeley)	• 100,000 driving videos over 1,100 hours • 100,000 images with bounding boxes for 10 categories • 100,000 images with lane markings • 100,000 images with drivable-area segmentation • 10,000 images with pixel-level instance segmentation
	Waymo Open Dataset	3,000 driving scenes totaling 16.7 hours of video data, 600,000 frames, approximately 25 million 3D bounding boxes, and 22 million 2D bounding boxes
Text	SQuAD	150,000 Question and Answer snippets from Wikipedia

	Yelp Reviews	Five million Yelp reviews
Satellite data	Landsat Data	Several million satellite images (100 nautical mile width and height), along with eight spectral bands (15- to 60-meter spatial resolution)
Audio	Google AudioSet	2,084,320 10-second sound clips from YouTube with 632 categories
	LibriSpeech	1,000 hours of read English speech

	Yelp Reviews	Five million Yelp reviews
Satellite data	Landsat Data	Several million satellite images (100 nautical mile width and height), along with eight spectral bands (15- to 60-meter spatial resolution)
Audio	Google AudioSet	2,084,320 10-second sound clips from YouTube with 632 categories
	LibriSpeech	1,000 hours of read English speech

Model Architecture

At a high level, a model is just a function. It takes in one or more inputs and gives an output. The input might be in the form of text, images, audio, video, and more. The output is a prediction. A good model is one whose predictions reliably match the expected reality. The model's accuracy on a dataset is a major determining factor as to whether it's suitable for use in a real-world application. For many people, this is all they really need to know about deep learning models. But it's when we peek into the inner workings of a model that it becomes really interesting (Figure 1-9).

Figure 1-9. A black box view of a deep learning model

Inside the model is a graph that consists of nodes and edges. Nodes represent mathematical operations, whereas edges represent how the data flows from one node to another. In other words, if the output of one node can become the input to one or more nodes, the connections between those nodes are represented by edges. The structure of this graph determines the potential for accuracy, its speed, how much resources it consumes (memory, compute, and energy), and the type of input it's capable of processing.

The layout of the nodes and edges is known as the *architecture* of the model. Essentially, it's a blueprint. Now, the blueprint is only half the picture. We still need the actual building. Training is the process that utilizes this blueprint to construct that building. We train a model by repeatedly 1) feeding it input data, 2) getting outputs from it, 3) monitoring how far these predictions are from the expected reality (i.e., the labels associated with the data), and then, 4) propagating the magnitude of error back to the model so that it can progressively learn to correct itself. This training process is performed iteratively until we are satisfied with the accuracy of the predictions.

The result from this training is a set of numbers (also known as weights) that is

assigned to each of the nodes. These weights are necessary parameters for the nodes in the graph to operate on the input given to them. Before the training begins, we usually assign random numbers as weights. The goal of the training process is essentially to gradually tune the values of each set of these weights until they, in conjunction with their corresponding nodes, produce satisfactory predictions.

To understand weights a little better, let's examine the following dataset with two inputs and one output:

Table 1-3. Example dataset

$input_1$	$input_2$	output
1	6	20
2	5	19
3	4	18
4	3	17
5	2	16
6	1	15

Using linear algebra (or guesswork in our minds), we can deduce that the equation governing this dataset is:

$$output = f(input_1, input_2) = 2 \text{ x } input_1 + 3 \text{ x } input_2$$

In this case, the weights for this mathematical operation are 2 and 3. A deep neural network has millions of such weight parameters.

Depending on the types of nodes used, different themes of model architectures will be better suited for different kinds of input data. For example, CNNs are used for image and audio, whereas Recurrent Neural Networks (RNNs) and LSTM are often used in text processing.

In general, training one of these models from scratch can take a pretty significan amount of time, potentially weeks. Luckily for us, many researchers have already done the difficult work of training them on a generic dataset (like ImageNet) and have made them available for everyone to use. What's even

better is that we can take these available models and tune them to our specific dataset. This process is called transfer learning and accounts for the vast majority of needs by practitioners.

Compared to training from scratch, transfer learning provides a two-fold advantage: significantly reduced training time a (few minutes to hours instead of weeks), and it can work with a substantially smaller dataset (hundreds to thousands of data samples instead of millions). Table 1-4 shows some famous examples of model architectures.

Table 1-4. Example model architectures over the years

Task	Example model architectures
Image classification	ResNet-152 (2015), MobileNet (2017)
Text classification	BERT (2018), XLNet (2019)
Image segmentation	U-Net (2015), DeepLabV3 (2018)
Image translation	Pix2Pix (2017)
Object detection	YOLO9000 (2016), Mask R-CNN (2017)
Speech generation	WaveNet (2016)

Each one of the models from Table 1-4 has a published accuracy metric on reference datasets (e.g., ImageNet for classification, MS COCO for detection). Additionally, these architectures have their own characteristic resource requirements (model size in megabytes, computation requirements in floating-point operations, or FLOPS).

We explore transfer learning in-depth in the upcoming chapters. Now, let's look at the kinds of deep learning frameworks and services that are available to us.

NOTE

t When Kaiming He et al. came up with the 152-layer ResNet architecture in 2015—a feat of its day considering the previous largest GoogLeNet model consisted of 22 layers—there was just one question on everyone's mind: "Why not 153 layers?" The reason, as it turns out, was that Kaiming ran out of GPU memory!

Frameworks

There are several deep learning libraries out there that help us train our models. Additionally, there are frameworks that specialize in using those trained models to make predictions (or *inference*), optimizing for where the application resides.

Historically, as is the case with software generally, many libraries have come and gone—Torch (2002), Theano (2007), Caffe (2013), Microsoft Cognitive Toolkit (2015), Caffe2 (2017)—and the landscape has been evolving rapidly. Learnings from each have made the other libraries easier to pick up, driven interest, and improved productivity for beginners and experts alike. Table 1-5 looks at some of the popular ones.

Table 1-5. Popular deep learning frameworks

Framework	Best suited for	Typical target platform
TensorFlow (including Keras)	Training	Desktops, servers
PyTorch	Training	Desktops, servers
MXNet	Training	Desktops, servers
TensorFlow Serving	Inference	Servers
TensorFlow Lite	Inference	Mobile and embedded devices
TensorFlow.js	Inference	Browsers
ml5.js	Inference	Browsers
Core ML	Inference	Apple devices
Xnor AI2GO	Inference	Embedded devices

TensorFlow

In 2011, Google Brain developed the DNN library DistBelief for internal research and engineering. It helped train Inception (2014's winning entry to the ImageNet Large Scale Visual Recognition Challenge) as well as helped improve the quality of speech recognition within Google products. Heavily tied into Google's infrastructure, it was not easy to configure and to share code with external machine learning enthusiasts. Realizing the limitations, Google began working on a second-generation distributed machine learning framework, which promised to be general-purpose, scalable, highly performant, and portable to

many hardware platforms. And the best part, it was open source. Google called i TensorFlow and announced its release on November 2015.

TensorFlow delivered on a lot of these aforementioned promises, developing an end-to-end ecosystem from development to deployment, and it gained a massive following in the process. With more than 100,000 stars on GitHub, it shows no signs of stopping. However, as adoption gained, users of the library rightly criticized it for not being easy enough to use. As the joke went, TensorFlow was a library by Google engineers, of Google engineers, for Google engineers, and if you were smart enough to use TensorFlow, you were smart enough to get hired there.

But Google was not alone here. Let's be honest. Even as late as 2015, it was a given that working with deep learning libraries would inevitably be an unpleasant experience. Forget even working on these; installing some of these frameworks made people want to pull their hair out. (Caffe users out there—doe this ring a bell?)

Keras

As an answer to the hardships faced by deep learning practitioners, François Chollet released the open source framework Keras in March 2015, and the worl hasn't been the same since. This solution suddenly made deep learning accessible to beginners. Keras provided an intuitive and easy-to-use interface fo coding, which would then use other deep learning libraries as the backend computational framework. Starting with Theano as its first backend, Keras encouraged rapid prototyping and reduced the number of lines of code. Eventually, this abstraction expanded to other frameworks including Cognitive Toolkit, MXNet, PlaidML, and, yes, TensorFlow.

PyTorch

In parallel, PyTorch started at Facebook early in 2016, where engineers had the benefit of observing TensorFlow's limitations. PyTorch supported native Python constructs and Python debugging right off the bat, making it flexible and easier to use, quickly becoming a favorite among AI researchers. It is the second-largest end-to-end deep learning system. Facebook additionally built Caffe2 to take PyTorch models and deploy them to production to serve more than a billion

users. Whereas PyTorch drove research, Caffe2 was primarily used in production. In 2018, Caffe2 was absorbed into PyTorch to make a full framework.

A continuously evolving landscape

Had this story ended with the ease of Keras and PyTorch, this book would not have the word “TensorFlow” in the subtitle. The TensorFlow team recognized that if it truly wanted to broaden the tool’s reach and democratize AI, it needed to make the tool easier. So it was welcome news when Keras was officially included as part of TensorFlow, offering the best of both worlds. This allowed developers to use Keras for defining the model and training it, and core TensorFlow for its high-performance data pipeline, including distributed training and ecosystem to deploy. It was a match made in heaven! And to top it all, TensorFlow 2.0 (released in 2019) included support for native Python constructs and eager execution, as we saw in PyTorch.

With so many competing frameworks available, the question of portability inevitability arises. Imagine a new research paper published with the state-of-the-art model being made public in PyTorch. If we didn’t work in PyTorch, we would be locked out of the research and would have to reimplement and train it. Developers like to be able to share models freely and not be restricted to a specific ecosystem. Organically, many developers wrote libraries to convert model formats from one library to another. It was a simple solution, except that i led to a combinatorial explosion of conversion tools that lacked official support and sufficient quality due to the sheer number of them. To address this issue, the Open Neural Network Exchange (ONNX) was championed by Microsoft and Facebook, along with major players in the industry. ONNX provided a specification for a common model format that was readable and writable by a number of popular libraries officially. Additionally, it provided converters for libraries that did not natively support this format. This allowed developers to train in one framework and do inferences in a different framework.

Apart from these frameworks, there are several Graphical User Interface (GUI) systems that make code-free training possible. Using transfer learning, they generate trained models quickly in several formats useful for inference. With point-and-click interfaces, even your grandma can now train a neural network!

Table 1-6. Popular GUI-based model training tools

Service	Platform
Microsoft CustomVision.AI	Web-based
Google AutoML	Web-based
Clarifai	Web-based
IBM Visual Recognition	Web-based
Apple Create ML	macOS
NVIDIA DIGITS	Desktop
Runway ML	Desktop

So why did we choose TensorFlow and Keras as the primary frameworks for this book? Considering the sheer amount of material available, including documentation, Stack Overflow answers, online courses, the vast community of contributors, platform and device support, industry adoption, and, yes, open jobs available (approximately three times as many TensorFlow-related roles compared to PyTorch in the United States), TensorFlow and Keras currently dominate the landscape when it comes to frameworks. It made sense for us to select this combination. That said, the techniques discussed in the book are generalizable to other libraries, as well. Picking up a new framework shouldn't take you too long. So, if you really want to move to a company that uses PyTorch exclusively, don't hesitate to apply there.

Table 1-6. Popular GUI-based model training tools

Service	Platform
Microsoft CustomVision.AI	Web-based
Google AutoML	Web-based
Clarifai	Web-based
IBM Visual Recognition	Web-based
Apple Create ML	macOS
NVIDIA DIGITS	Desktop
Runway ML	Desktop

So why did we choose TensorFlow and Keras as the primary frameworks for this book? Considering the sheer amount of material available, including documentation, Stack Overflow answers, online courses, the vast community of contributors, platform and device support, industry adoption, and, yes, open jobs available (approximately three times as many TensorFlow-related roles compared to PyTorch in the United States), TensorFlow and Keras currently dominate the landscape when it comes to frameworks. It made sense for us to select this combination. That said, the techniques discussed in the book are generalizable to other libraries, as well. Picking up a new framework shouldn’t take you too long. So, if you really want to move to a company that uses PyTorch exclusively, don’t hesitate to apply there.

Hardware

In 1848, when James W. Marshall discovered gold in California, the news spread like wildfire across the United States. Hundreds of thousands of people stormed to the state to begin mining for riches. This was known as the *California Gold Rush*. Early movers were able to extract a decent chunk, but the latecomers were not nearly as lucky. But the rush did not stop for many years. Can you guess who made the most money throughout this period? The shovel makers!

Cloud and hardware companies are the shovel makers of the twenty-first century. Don't believe us? Look at the stock performance of Microsoft and NVIDIA in the past decade. The only difference between 1849 and now is the mind-bogglingly large amount of shovel choices available to us.

Given the variety of hardware available, it is important to make the correct choices for the constraints imposed by resource, latency, budget, privacy, and legal requirements of the application.

Depending on how your application interacts with the user, the inference phase usually has a user waiting at the other end for a response. This imposes restrictions on the type of hardware that can be used as well as the location of the hardware. For example, a Snapchat lens cannot run on the cloud due to the network latency issues. Additionally, it needs to run in close to real time to provide a good user experience (UX), thus setting a minimum requirement on the number of frames processed per second (typically >15 fps). On the other hand, a photo uploaded to an image library such as Google Photos does not need immediate image categorization done on it. A few seconds or few minutes of latency is acceptable.

Going to the other extreme, training takes a lot more time; anywhere between minutes to hours to days. Depending on our training scenario, the real value of better hardware is enabling faster experimentation and more iterations. For anything more serious than basic neural networks, better hardware can make a mountain of difference. Typically, GPUs would speed things up by 10 to 15 times compared to CPUs, and at a much higher performance per watt, reducing the wait time for our experiment to finish from a week to a few hours. This can be the difference in watching a documentary about the Grand Canyon (two

hours) versus actually making the trip to visit the Grand Canyon (four days).

Following are a few fundamental hardware categories to choose from and how they are typically characterized (see also Figure 1-10):

Central Processing Unit (CPU)

Cheap, flexible, slow. For example, Intel Core i9-9900K.

GPU

High throughput, great for batching to utilize parallel processing, expensive. For example, NVIDIA GeForce RTX 2080 Ti.

Field-Programmable Gate Array (FPGA)

Fast, low power, reprogrammable for custom solutions, expensive. Known companies include Xilinx, Lattice Semiconductor, Altera (Intel). Because of the ability to run in seconds and configurability to any AI model, Microsoft Bing runs the majority of its AI on FPGAs.

Application-Specific Integrated Circuit (ASIC)

Custom-made chip. Extremely expensive to design, but inexpensive when built for scale. Just like in the pharmaceutical industry, the first item costs the most due to the R&D effort that goes into designing and making it. Producing massive quantities is rather inexpensive. Specific examples include the following:

Tensor Processing Unit (TPU)

ASIC specializing in operations for neural networks, available on Google Cloud only.

Edge TPU

Smaller than a US penny, accelerates inference on the edge.

Neural Processing Unit (NPU)

Often used by smartphone manufacturers, this is a dedicated chip for accelerating neural network inference.

Figure 1-10. Comparison of different types of hardware relative to flexibility, performance, and cost

Let's look at a few scenarios for which each one would be used:

- Getting started with training → CPU
- Training large networks → GPUs and TPUs
- Inference on smartphones → Mobile CPU, GPU, Digital Signal Processor (DSP), NPU
- Wearables (e.g., smart glasses, smartwatches) → Edge TPU, NPUs
- Embedded AI projects (e.g., flood surveying drone, autonomous wheelchair) → Accelerators like Google Coral, Intel Movidius with Raspberry Pi, or GPUs like NVIDIA Jetson Nano, all the way down to $15 microcontrollers (MCUs) for wake word detection in smart speakers

As we go through the book, we will closely explore many of these.

Figure 1-10. Comparison of different types of hardware relative to flexibility, performance, and cost

Let's look at a few scenarios for which each one would be used:

- Getting started with training → CPU
- Training large networks → GPUs and TPUs
- Inference on smartphones → Mobile CPU, GPU, Digital Signal Processor (DSP), NPU
- Wearables (e.g., smart glasses, smartwatches) → Edge TPU, NPUs
- Embedded AI projects (e.g., flood surveying drone, autonomous wheelchair) → Accelerators like Google Coral, Intel Movidius with Raspberry Pi, or GPUs like NVIDIA Jetson Nano, all the way down to $15 microcontrollers (MCUs) for wake word detection in smart speakers

As we go through the book, we will closely explore many of these.

Responsible AI

So far, we have explored the power and the potential of AI. It shows great promise to enhance our abilities, to make us more productive, to give us superpowers.

But with great power comes great responsibility.

As much as AI can help humanity, it also has equal potential to harm us when not designed with thought and care (either intentionally or unintentionally). The AI is not to blame; rather, it's the AI's designers.

Consider some real incidents that made the news in the past few years.

- "____ can allegedly determine whether you're a terrorist just by analyzing your face" (Figure 1-11): *Computer World*, 2016
- "AI is sending people to jail—and getting it wrong": *MIT Tech Review*, 2019
- "____ supercomputer recommended 'unsafe and incorrect' cancer treatments, internal documents show": *STAT News*, 2018
- "____ built an AI tool to hire people but had to shut it down because it was discriminating against women": *Business Insider*, 2018
- "____ AI study: Major object recognition systems favor people with more money": *VentureBeat*, 2019
- "____ labeled black people 'gorillas'" *USA Today*, 2015. "Two years later, ____ solves 'racist algorithm' problem by purging 'gorilla' label from image classifier": *Boing Boing*, 2018
- "____ silences its new A.I. bot Tay, after Twitter users teach it racism": *TechCrunch*, 2016
- "AI Mistakes Bus-Side Ad for Famous CEO, Charges Her With Jaywalking": *Caixin Global*, 2018
- "____ to drop Pentagon AI contract after employee objections to the

‘business of war’”: *Washington Post*, 2018

- “Self-driving ____ death car ‘spotted pedestrian six seconds before mowing down and killing her’”: *The Sun*, 2018

Figure 1-11. Startup claiming to classify people based on their facial structure

Can you fill in the blanks here? We’ll give you some options—Amazon, Microsoft, Google, IBM, and Uber. Go ahead and fill them out. We’ll wait.

There’s a reason we kept them blank. It’s to recognize that it’s not a problem belonging to a specific individual or a company. This is everyone’s problem. And although these things happened in the past, and might not reflect the current state, we can learn from them and try not to make the same mistakes. The silver lining here is that everyone learned from these mistakes.

We, as developers, designers, architects, and leaders of AI, have the responsibility to think beyond just the technical problem at face value. Following are just a handful of topics that are relevant to any problem we solve (AI or otherwise). They must not take a backseat.

‘business of war’”: *Washington Post*, 2018

- “Self-driving ____ death car ‘spotted pedestrian six seconds before mowing down and killing her’”: *The Sun*, 2018

Figure 1-11. Startup claiming to classify people based on their facial structure

Can you fill in the blanks here? We’ll give you some options—Amazon, Microsoft, Google, IBM, and Uber. Go ahead and fill them out. We’ll wait.

There’s a reason we kept them blank. It’s to recognize that it’s not a problem belonging to a specific individual or a company. This is everyone’s problem. And although these things happened in the past, and might not reflect the current state, we can learn from them and try not to make the same mistakes. The silver lining here is that everyone learned from these mistakes.

We, as developers, designers, architects, and leaders of AI, have the responsibility to think beyond just the technical problem at face value. Following are just a handful of topics that are relevant to any problem we solve (AI or otherwise). They must not take a backseat.

Bias

Often in our everyday work, we bring in our own biases, knowingly or unknowingly. This is the result of a multitude of factors including our environment, upbringing, cultural norms, and even our inherent nature. After all, AI and the datasets that power them were not created in a vacuum—they were created by human beings with their own biases. Computers don't magically create bias on their own, they reflect and amplify existing ones.

Take the example from the early days of the YouTube app when the developers noticed that roughly 10% of uploaded videos were upside-down. Maybe if that number had been lower, say 1%, it could have been brushed off as user error. Bu 10% was too high a number to be ignored. Do you know who happens to make up 10% of the population? Left-handed people! These users were holding their phones in the opposite orientation as their right-handed peers. But the engineers at YouTube had not accounted for that case during the development and testing of their mobile app, so YouTube uploaded videos to its server in the same orientation for both left-handed and right-handed users.

This problem could have been caught much earlier if the developers had even a single left-handed person on the team. This simple example demonstrates the importance of diversity. Handedness is just one small attribute that defines an individual. Numerous other factors, often outside their control, often come into play. Factors such as gender, skin tone, economic status, disability, country of origin, speech patterns, or even something as trivial as hair length can determine life-changing outcomes for someone, including how an algorithm treats them.

Google's machine learning glossary lists several forms of bias that can affect a machine learning pipeline. The following are just some of them:

Selection bias

> The dataset is not representative of the distribution of the real-world problem and is skewed toward a subset of categories. For example, in many virtual assistants and smart home speakers, some spoken accents are overrepresented, whereas other accents have no data at all in the training dataset, resulting in a poor UX for large chunks of the world's population.

Selection bias can also happen because of co-occurrence of concepts. For example, Google Translate, when used to translate the sentences “She is a doctor. He is a nurse” into a gender-neutral language such as Turkish and then back, switches the genders, as demonstrated in Figure 1-12. This is likely because the dataset contains a large sample of co-occurrences of male pronouns and the word “doctor,” and female pronouns and the word “nurse.”

Figure 1-12. Google Translate reflecting the underlying bias in data (as of September 2019)

Implicit bias

This type of bias creeps in because of implicit assumptions that we all make when we see something. Consider the highlighted portion in Figure 1-13. Anyone shown it might assume with a high amount of certainty that those stripes belong to a zebra. In fact, given how much ImageNet-trained networks are biased toward textures,[2] most of them will classify the full image as a zebra. Except that we know that the image is of a sofa upholstered in a zebra-like fabric.

Figure 1-13. Zebra sofa by Glen Edelson (image source)

Reporting bias

Sometimes the loudest voices in the room are the most extreme ones and dominate the conversation. One good look at Twitter might make it seem as if the world is ending, whereas most people are busy leading mundane lives. Unfortunately, boring does not sell.

In-group/out-group bias

An annotator from East Asia might look at a picture of the Statue of Liberty and give it tags like “America” or “United States,” whereas someone from the US might look at the same picture and assign more granular tags such as “New York” or “Liberty Island.” It’s human nature to see one’s own groups with nuance while seeing other groups as more homogenous, and that reflects in our datasets, as well.

Accountability and Explainability

Imagine, in the late 1800s, Mr. Karl Benz told you that he invented this four-wheeled device that could transport you quicker than anything else in existence. Except he had no idea how it worked. All he knew was that it consumed a highly flammable liquid that exploded several times inside it to propel it forward. What caused it to move? What caused it to stop? What stopped it from burning the person sitting inside it? He had no answers. If this was the origin story of the car you'd probably not want to get into that contraption.

This is precisely what is happening with AI right now. Previously, with traditional machine learning, data scientists had to manually pick features (predictive variables) from data, from which a machine learning model then would learn. This manual selection process, although cumbersome and restrictive, gave them more control and insight into how the prediction came about. However, with deep learning, these features are automatically selected. Data scientists are able to build models by providing lots of data, and these models somehow end up making predictions reliably—most of the time. But the data scientist doesn't know exactly how the model works, what features it learned, under what circumstances the model works, and, more importantly, the circumstances under which it doesn't work. This approach might be acceptable when Netflix is recommending TV shows to us based on what we've already watched (although we're fairly certain they have the line `recommendations.append("Stranger Things")` in their code somewhere). But AI does a lot more than just recommend movies these days. Police and judicial systems are beginning to rely on algorithms to decide whether someone poses a risk to society and whether they should be detained before their trial. The lives and freedoms of many people are at stake. We simply must not outsource important decision making to an unaccountable black box. Thankfully, there's momentum to change that with investments in *Explainable AI*, wherein the model would be able to not just provide predictions but also account for the factors that caused it to make a certain prediction, and reveal areas of limitations

Additionally, cities (such as New York) are beginning to make their algorithms accountable to the public by recognizing that the public has a right to know what algorithms they use for vital decision making and how they work, allowing

reviews and audits by experts, improving expertise in government agencies to better evaluate each system they add, and by providing mechanisms to dispute a decision made by an algorithm.

reviews and audits by experts, improving expertise in government agencies to better evaluate each system they add, and by providing mechanisms to dispute a decision made by an algorithm.

Reproducibility

Research performed in the scientific field gains wide acceptance by the community only when it's reproducible; that is, anyone studying the research should be able to replicate the conditions of the test and obtain the same results. Unless we can reproduce a model's past results, we cannot hold it accountable when using it in the future. In the absence of reproducibility, research is vulnerable to *p-hacking*—tweaking the parameters of an experiment until the desired results are obtained. It's vital for researchers to extensively document their experimental conditions, including the dataset(s), benchmarks, and algorithms, and declare the hypothesis they will be testing prior to performing an experiment. Trust in institutions is at an all-time low and research that is not grounded in reality, yet sensationalized by the media, can erode that trust even more. Traditionally, replicating a research paper was considered a dark art because many implementation details are left out. The uplifting news is that researchers are now gradually beginning to use publicly available benchmarks (as opposed to their privately constructed datasets) and open sourcing the code they used for their research. Members of the community can piggyback on this code, prove it works, and make it better, thereby leading to newer innovations rapidly.

Reproducibility

Research performed in the scientific field gains wide acceptance by the community only when it's reproducible; that is, anyone studying the research should be able to replicate the conditions of the test and obtain the same results. Unless we can reproduce a model's past results, we cannot hold it accountable when using it in the future. In the absence of reproducibility, research is vulnerable to *p-hacking*—tweaking the parameters of an experiment until the desired results are obtained. It's vital for researchers to extensively document their experimental conditions, including the dataset(s), benchmarks, and algorithms, and declare the hypothesis they will be testing prior to performing an experiment. Trust in institutions is at an all-time low and research that is not grounded in reality, yet sensationalized by the media, can erode that trust even more. Traditionally, replicating a research paper was considered a dark art because many implementation details are left out. The uplifting news is that researchers are now gradually beginning to use publicly available benchmarks (as opposed to their privately constructed datasets) and open sourcing the code they used for their research. Members of the community can piggyback on this code, prove it works, and make it better, thereby leading to newer innovations rapidly.

Robustness

There’s an entire area of research on one-pixel attacks on CNNs. Essentially, the objective is to find and modify a single pixel in an image to make a CNN predict something entirely different. For example, changing a single pixel in a picture of an apple might result in a CNN classifying it as a dog. A lot of other factors can influence predictions, such as noise, lighting conditions, camera angle, and more that would not have affected a human’s ability to make a similar call. This is particularly relevant for self-driving cars, where it would be possible for a bad actor on a street to modify the input the car sees in order to manipulate it into doing bad things. In fact, Tencent’s Keen Security Lab was able to exploit a vulnerability in Tesla’s AutoPilot by strategically placing small stickers on the road, which led it to change lanes and drive into the oncoming lane. Robust AI that is capable of withstanding noise, slight deviations, and intentional manipulation is necessary if we are to be able to trust it.

Robustness

There’s an entire area of research on one-pixel attacks on CNNs. Essentially, the objective is to find and modify a single pixel in an image to make a CNN predict something entirely different. For example, changing a single pixel in a picture of an apple might result in a CNN classifying it as a dog. A lot of other factors can influence predictions, such as noise, lighting conditions, camera angle, and more that would not have affected a human’s ability to make a similar call. This is particularly relevant for self-driving cars, where it would be possible for a bad actor on a street to modify the input the car sees in order to manipulate it into doing bad things. In fact, Tencent’s Keen Security Lab was able to exploit a vulnerability in Tesla’s AutoPilot by strategically placing small stickers on the road, which led it to change lanes and drive into the oncoming lane. Robust AI that is capable of withstanding noise, slight deviations, and intentional manipulation is necessary if we are to be able to trust it.

Privacy

In the pursuit of building better and better AI, businesses need to collect lots of data. Unfortunately, sometimes they overstep their bounds and collect information overzealously beyond what is necessary for the task at hand. A business might believe that it is using the data it collects only for good. But what if it is acquired by a company that does not have the same ethical boundaries for data use? The consumer's information could be used for purposes beyond the originally intended goals. Additionally, all that data collected in one place makes it an attractive target for hackers, who steal personal information and sell it on the black market to criminal enterprises. Moreover, governments are already overreaching in an attempt to track each and every individual.

All of this is at odds to the universally recognized human right of privacy. What consumers desire is having transparency into what data is being collected about them, who has access to it, how it's being used, and mechanisms to opt out of the data collection process, as well to delete data that was already collected on them

As developers, we want to be aware of all the data we are collecting, and ask ourselves whether a piece of data is even necessary to be collected in the first place. To minimize the data we collect, we could implement privacy-aware machine learning techniques such as Federated Learning (used in Google Keyboard) that allow us to train networks on the users' devices without having to send any of the Personally Identifiable Information (PII) to a server.

It turns out that in many of the aforementioned headlines at the beginning of this section, it was the bad PR fallout that brought mainstream awareness of these topics, introduced accountability, and caused an industry-wide shift in mindset to prevent repeats in the future. We must continue to hold ourselves, academics, industry leaders, and politicians accountable at every misstep and act swiftly to fix the wrongs. Every decision we make and every action we take has the potential to set a precedent for decades to come. As AI becomes ubiquitous, we need to come together to ask the tough questions and find answers for them if we want to minimize the potential harm while reaping the maximum benefits.

t

s

e

.

o

o

e

Summary

This chapter explored the landscape of the exciting world of AI and deep learning. We traced the timeline of AI from its humble origins, periods of great promise, through the dark AI winters, and up to its present-day resurgence. Along the way, we answered the question of why it's different this time. We then looked at the necessary ingredients to build a deep learning solution, including datasets, model architectures, frameworks, and hardware. This sets us up for further exploration in the upcoming chapters. We hope you enjoy the rest of the book. It's time to dig in!

Summary

This chapter explored the landscape of the exciting world of AI and deep learning. We traced the timeline of AI from its humble origins, periods of great promise, through the dark AI winters, and up to its present-day resurgence. Along the way, we answered the question of why it's different this time. We then looked at the necessary ingredients to build a deep learning solution, including datasets, model architectures, frameworks, and hardware. This sets us up for further exploration in the upcoming chapters. We hope you enjoy the rest of the book. It's time to dig in!

Frequently Asked Questions

1. I'm just getting started. Do I need to spend a lot of money on buying powerful hardware?

   Luckily for you, you can get started even with your web browser. All of our scripts are available online, and can be run on free GPUs courtesy of the kind people at Google Colab (Figure 1-14), who are generously making powerful GPUs available for free (up to 12 hours at a time). This should get you started. As you become better at it by performing more experiments (especially in a professional capacity or on large datasets), you might want to get a GPU either by renting one on the cloud (Microsoft Azure, Amazon Web Services (AWS), Google Cloud Platform (GCP), and others) or purchasing the hardware. Watch out for those electricity bills, though!

Figure 1-14. Screenshot of a notebook on GitHub running on Colab inside Chrome

2. Colab is great, but I already have a powerful computer that I purchased for playing <insert name of video game>. How should I set up my environment?

   The ideal setup involves Linux, but Windows and macOS work, too. For

most chapters, you need the following:

- Python 3 and PIP
- `tensorflow` or `tensorflow-gpu` PIP package (version 2 or greater)
- Pillow

We like keeping things clean and self-contained, so we recommend using Python virtual environments. You should use the virtual environment whenever you install a package or run a script or a notebook.

If you do not have a GPU, you are done with the setup.

If you have an NVIDIA GPU, you would want to install the appropriate drivers, then CUDA, then cuDNN, then `tensorflow-gpu` package. If you're using Ubuntu, there's an easier solution than installing these packages manually, which can be tedious and error prone even for the best of us: simply install the entire environment with just one line using Lambda Stack.

Alternatively, you could install all of your packages using Anaconda Distribution, which works equally well for Windows, Mac, and Linux.

3. Where will I find the code used in this book?

   You'll find ready-to-run examples at *http://PracticalDeepLearning.ai*.

4. What are the minimal prerequisites to be able to read this book?

   A Ph.D. in areas including Calculus, Statistical Analysis, Variational Autoencoders, Operations Research, and so on...are definitely *not* necessary to be able to read this book (had you a little nervous, didn't we?). Some basic coding skills, familiarity with Python, a healthy amount of curiosity, and a sense of humor should go a long way in the process of absorbing the material. Although a beginner-level understanding of mobile development (with Swift and/or Kotlin) will help, we've designed the examples to be self-sufficient and easy enough to be deployed by someone who has never written a mobile app previously.

5. What frameworks will we be using?

Keras + TensorFlow for training. And chapter by chapter, we explore different inference frameworks.

6. Will I be an expert when I finish this book?

   If you follow along, you'll have the know-how on a wide variety of topics all the way from training to inference, to maximizing performance. Even though this book primarily focuses on computer vision, you can bring the same know-how to other areas such as text, audio, and so on and get up to speed very quickly.

7. Who is the cat from earlier in the chapter?

   That is Meher's cat, Vader. He will be making multiple cameos throughout this book. And don't worry, he has already signed a model release form.

8. Can I contact you?

   Sure. Drop us an email at PracticalDLBook@gmail.com with any questions, corrections, or whatever, or tweet to us @PracticalDLBook.

---

1 If you're reading a pirated copy, consider us disappointed in you.

2 Robert Geirhos et al.

Keras + TensorFlow for training. And chapter by chapter, we explore different inference frameworks.

6. Will I be an expert when I finish this book?

   If you follow along, you'll have the know-how on a wide variety of topics all the way from training to inference, to maximizing performance. Even though this book primarily focuses on computer vision, you can bring the same know-how to other areas such as text, audio, and so on and get up to speed very quickly.

7. Who is the cat from earlier in the chapter?

   That is Meher's cat, Vader. He will be making multiple cameos throughout this book. And don't worry, he has already signed a model release form.

8. Can I contact you?

   Sure. Drop us an email at PracticalDLBook@gmail.com with any questions, corrections, or whatever, or tweet to us @PracticalDLBook.

---

1 If you're reading a pirated copy, consider us disappointed in you.

2 Robert Geirhos et al.

Chapter 2. What's in the Picture: Image Classification with Keras

If you have skimmed through deep learning literature, you might have come across a barrage of academic explanations laced with intimidating mathematics. Don't worry. We will ease you into practical deep learning with an example of classifying images with just a few lines of code.

In this chapter, we take a closer look at the Keras framework, discuss its place in the deep learning landscape, and then use it to classify a few images using existing state-of-the-art classifiers. We visually investigate how these classifiers operate by using *heatmaps*. With these heatmaps, we make a fun project in which we classify objects in videos.

Recall from the "Recipe for the Perfect Deep Learning Solution" that we need four ingredients to create our deep learning recipe: hardware, dataset, framework, and model. Let's see how each of these comes into play in this chapter:

- We begin with the easy one: *hardware*. Even an inexpensive laptop would suffice for what we we're doing in this chapter. Alternatively, you can run the code in this chapter by opening the GitHub notebook (see *http://PracticalDeepLearning.ai*) in Colab. This is just a matter of a few mouse clicks.
- Because we won't be training a neural network just yet, we don't need a *dataset* (other than a handful of sample photos to test with).
- Next, we come to the *framework*. This chapter's title has Keras in it, so that is what we will be using for now. In fact, we use Keras for our training needs throughout a good part of the book.
- One way to approach a deep learning problem is to obtain a dataset, write the code to train it, spend a lot of time and energy (both human and electrical) in training that model, and then use it for making predictions. Bu

we are not gluttons for punishment. So, we will use a *pretrained model* instead. After all, the research community has already spent blood, sweat, and tears training and publishing many of the standard models that are now publicly available. We will be reusing one of the more famous models called ResNet-50, the little sibling of ResNet-152 that won the ILSVRC in 2015.

You will get hands-on with some code in this chapter. As we all know, the best way to learn is by doing. You might be wondering, though, what's the theory behind this? That comes in later chapters, in which we delve deeper into the nuts and bolts of CNNs using this chapter as a foundation.

we are not gluttons for punishment. So, we will use a *pretrained model* instead. After all, the research community has already spent blood, sweat, and tears training and publishing many of the standard models that are now publicly available. We will be reusing one of the more famous models called ResNet-50, the little sibling of ResNet-152 that won the ILSVRC in 2015.

You will get hands-on with some code in this chapter. As we all know, the best way to learn is by doing. You might be wondering, though, what's the theory behind this? That comes in later chapters, in which we delve deeper into the nuts and bolts of CNNs using this chapter as a foundation.

Introducing Keras

As Chapter 1 discussed, Keras started in 2015 as an easy-to-use abstraction layer over other libraries, making rapid prototyping possible. This made the learning curve a lot less steep for beginners of deep learning. At the same time, it made deep learning experts more productive by helping them rapidly iterate on experiments. In fact, the majority of the winning teams on *Kaggle.com* (which hosts data science competitions) have used Keras. Eventually, in 2017, the full implementation of Keras was available directly in TensorFlow, thereby combining the high scalability, performance, and vast ecosystem of TensorFlow with the ease of Keras. On the web, we often see the TensorFlow version of Keras referred to as `tf.keras`.

In this chapter and Chapter 3, we write all of the code exclusively in Keras. That includes boilerplate functions such as file reading, image manipulation (augmentation), and so on. We do this primarily for ease of learning. From Chapter 5 onward, we begin to gradually use more of the native performant TensorFlow functions directly for more configurability and control.

FROM THE CREATOR'S DESK

By François Chollet, creator of Keras, AI researcher, and author of *Deep Learning with Python*

I originally started Keras for my own use. At the time, in late 2014 and early 2015, there weren't any good options for deep learning frameworks with solid usability and strong support for both RNNs and convnets. Back then, I wasn't deliberately trying to democratize deep learning, I was just building what I wanted for myself. But as time went by, I saw lots of people pick up deep learning through Keras, and use it to solve many different problems I didn't even know existed. To me, that has been really fascinating. It made me realize that deep learning can be deployed, in transformative ways, to far more domains than what machine learning researchers are aware of. There are so many people out there that could benefit from using these technologies in their work. Because of that, I've grown to care a lot about making deep learning accessible to as many people as possible. That's the

only way we're going to deploy AI to the full extent of its potential—by making it broadly available. Today, in TensorFlow 2.0, the Keras API consolidates the power of deep learning in a spectrum of really productive and enjoyable workflows, suited to a variety of user profiles, from research to applications, including deployment. I'm looking forward to seeing what you'll build with it!

only way we’re going to deploy AI to the full extent of its potential—by making it broadly available. Today, in TensorFlow 2.0, the Keras API consolidates the power of deep learning in a spectrum of really productive and enjoyable workflows, suited to a variety of user profiles, from research to applications, including deployment. I’m looking forward to seeing what you’ll build with it!

Predicting an Image's Category

In layperson's terms, image classification answers the question: "what object does this image contain?" More specifically, "This image contains *X* object with what probability," where *X* is from a predefined list of categories of objects. If the probability is higher than a minimum threshold, the image is likely to contain one or more instances of *X*.

A simple image classification pipeline would consist of the following steps:

1. Load an image.
2. Resize it to a predefined size such as 224 x 224 pixels.
3. Scale the values of the pixel to the range [0,1] or [–1,1], a.k.a. normalization.
4. Select a pretrained model.
5. Run the pretrained model on the image to get a list of category predictions and their respective probabilities.
6. Display a few of the highest probability categories.

TIP

The GitHub link is provided on the website *http://PracticalDeepLearning.ai*. Navigate to `code/chapter-2` where you will find the Jupyter notebook `1-predict-class.ipynb` that has all the steps detailed.

We begin by importing all of the necessary modules from the Keras and Python packages:

```
import tensorflow as tf
from tf.keras.applications.resnet50 import preprocess_input, decode_predictions
from tf.keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
```

Next, we load and display the image that we want to classify (see Figure 2-1):

```
img_path = "../../sample-images/cat.jpg"
img = image.load_img(img_path, target_size=(224, 224))
plt.imshow(img)
plt.show()
```

Figure 2-1. Plot showing the contents of the input file

Yup, it's a cat (although the filename kind of gave it away). And that's what our model should ideally be predicting.

A BRIEF REFRESHER ON IMAGES

Before we dive into how images are processed, it would be good to take a look at how images store information. At the most basic level, an image is a collection of pixels that are laid out in a rectangular grid. Depending on the type of image, each pixel can consist of 1 to 4 parts (also known as *components* or *channels*). For the images we're using, these components represent the intensities of the colors red, green, and blue (RGB). They are typically 8 bits in length, so their values range between 0 and 255 (i.e., $2^8 - 1$).

Before feeding any image to Keras, we want to convert it to a standard format.

This is because pretrained models expect the input to be of a specific size. The standardization in our case involves resizing the image to 224 x 224 pixels.

Most deep learning models expect a batch of images as input. But what do we do when we have just one image? We create a batch of one image, of course! That essentially involves making an array consisting of that one object. Another way to look at this is to expand the number of dimensions from three (representing the three channels of the image) to four (the extra one for the length of the array itself).

If that is not clear, consider this scenario: for a batch of 64 images of size 224 x 224 pixels, each containing three channels (RGB), the object representing that batch would have a shape 64 x 224 x 224 x 3. In the code that follows, where we'd be using only one 224 x 224 x 3 image, we'd create a batch of just that image by expanding the dimensions from three to four. The shape of this newly created batch would be 1 x 224 x 224 x 3:

```
img_array = image.img_to_array(img)
img_batch = np.expand_dims(img_array, axis=0) # Increase the number of dimensions
```

In machine learning, models perform best when they are fed with data within a consistent range. Ranges typically include [0,1] and [–1,1]. Given that image pixel values are between 0 and 255, running the `preprocess_input` function from Keras on input images will normalize each pixel to a standard range. *Normalization* or *feature scaling* is one of the core steps in preprocessing images to make them suitable for deep learning.

Now comes the model. We will be using a *Convolutional Neural Network* (CNN) called ResNet-50. The very first question we should ask is, "Where will I find the model?" Of course, we could hunt for it on the internet to find something that is compatible with our deep learning framework (Keras). *But ain't nobody got time for that!* Luckily, Keras loves to make things easy and provides it to us in a single function call. After we call this function for the first time, the model will be downloaded from a remote server and cached locally:

```
model = tf.keras.applications.resnet50.ResNet50()
```

When predicting with this model, the results include probability predictions for

each class. Keras also provides the `decode_predictions` function, which tells us the probability of each category of objects contained in the image.

Now, let's see the entire code in one handy function:

```
def classify(img_path):
    img = image.load_img(img_path, target_size=(224, 224))
    model = tf.keras.applications.resnet50.ResNet50()
    img_array = image.img_to_array(img)
    img_batch = np.expand_dims(img_array, axis=0)
    img_preprocessed = preprocess_input(img_batch)
    prediction = model.predict(img_preprocessed)
    print(decode_predictions(prediction, top=3)[0])

classify("../../sample-images/cat.jpg")
```

```
[('n02123045', 'tabby', 0.50009364),
 ('n02124075', 'Egyptian_cat', 0.21690978),
 ('n02123159', 'tiger_cat', 0.2061722)]
```

The predicted categories for this image are various types of felines. Why doesn't it simply predict the word "cat," instead? The short answer is that the ResNet-50 model was trained on a granular dataset with many categories and does not include the more general "cat." We investigate this dataset in more detail a little later, but first, let's load another sample image (see Figure 2-2):

```
img_path = '../../sample-images/dog.jpg'
img = image.load_img(img_path, target_size=(224, 224))
plt.imshow(img)
plt.show()
```

Figure 2-2. Plot showing the contents of the file dog.jpg

And, again, we run our handy function from earlier:

```
classify("../../sample-images/dog.jpg")
```

```
[(u'n02113186', u'Cardigan', 0.809839),
 (u'n02113023', u'Pembroke', 0.17665945),
 (u'n02110806', u'basenji', 0.0042166105)]
```

As expected, we get different breeds of canines (and not just the “dog” category). If you are unfamiliar with the Corgi breed of dogs, the word “corgi” literally means “dwarf dog” in Welsh. The Cardigan and Pembroke are subbreeds of the Corgi family, which happen to look pretty similar to one another. It’s no wonder our model thinks that way, too.

Notice the predicted probability of each category. Usually, the prediction with the highest probability is considered the answer. Alternatively, any value over a predefined threshold can be considered as the answer, too. In the dog example, if we set a threshold of 0.5, Cardigan would be our answer.

Figure 2-3. Running the notebook on Google Colab using the browser

TIP

You can follow along with the code in this chapter and execute it interactively without any installations in the browser itself with Google Colab. Simply find the “Run on Colab” link at the top of each notebook on GitHub that you’d like to experiment with. Then, click the “Run Cell” button; this should execute the code within that cell, as shown in Figure 2-3.

f

Figure 2-3. Running the notebook on Google Colab using the browser

TIP

You can follow along with the code in this chapter and execute it interactively without any installations in the browser itself with Google Colab. Simply find the “Run on Colab” link at the top of each notebook on GitHub that you’d like to experiment with. Then, click the “Run Cell” button; this should execute the code within that cell, as shown in Figure 2-3.

Investigating the Model

We got the predictions from our model, great! But what factors led to those predictions? There are a few questions that we need to ask here:

- What dataset was the model trained on?
- Are there other models that I can use? How good are they? Where can I get them?
- Why does my model predict what it predicts?

We look into the answers to each of these questions in this section.

Investigating the Model

We got the predictions from our model, great! But what factors led to those predictions? There are a few questions that we need to ask here:

- What dataset was the model trained on?
- Are there other models that I can use? How good are they? Where can I get them?
- Why does my model predict what it predicts?

We look into the answers to each of these questions in this section.

ImageNet Dataset

Let's investigate the ImageNet dataset on which ResNet-50 was trained. ImageNet, as the name suggests, is a network of images; that is, a dataset of images organized as a network, as demonstrated in Figure 2-4. It is arranged in a hierarchical manner (like the WordNet hierarchy) such that the parent node encompasses a collection of images of all different varieties possible within that parent. For example, within the "animal" parent node, there are fish, birds, mammals, invertebrates, and so on. Each category has multiple subcategories, and these have subsubcategories, and so forth. For example, the category "American water spaniel" is eight levels from the root. The dog category contains 189 total subcategories in five hierarchical levels.

Visually, we developed the tree diagram shown in Figure 2-5 to help you to understand the wide variety of high-level entities that the ImageNet dataset contains. This treemap also shows the relative percentage of different categories that make up the ImageNet dataset.

Figure 2-4. The categories and subcategories in the ImageNet dataset

Figure 2-5. Treemap of ImageNet and its classes

The ImageNet dataset was the basis for the famous ILSVRC that started in 2010

to benchmark progress in computer vision and challenge researchers to innovate on tasks including object classification. Recall from Chapter 1 that the ImageNet challenge saw submissions that drastically improved in accuracy each year. When it started out, the error rate was nearly 30%. And now, it is 2.2%, already better than how an average human would perform at this task. This dataset and challenge are considered the single biggest reasons for the recent advancements in computer vision.

Wait, AI has better-than-human accuracy? If the dataset was created by humans, won't humans have 100% accuracy? Well, the dataset was created by experts, with each image verified by multiple people. Then Stanford researcher (and now of Tesla fame) Andrej Karpathy attempted to figure out how much a normal human would fare on ImageNet-1000. Turns out he achieved an accuracy of 94.9%, well short of the 100% we all expected. Andrej painstakingly spent a week going over 1,500 images, spending approximately one minute per image in tagging it. How did he misclassify 5.1% of the images? The reasons are a bit subtle:

Fine-grained recognition

For many people, it is really tough to distinguish a Siberian husky from a Alaskan Malamute. Someone who is really familiar with dog breeds would be able to tell them apart because they look for finer-level details that distinguish both breeds. It turns out that neural networks are capable of learning those finer-level details much more easily than humans.

Category unawareness

Not everyone is aware of all the 120 breeds of dogs and most certainly not each one of the 1,000 classes. But the AI is. After all, it was trained on it.

NOTE

Similar to ImageNet, speech datasets like Switchboard report a 5.1% error rate for speech transcription (coincidentally the same number as ImageNet). It's clear that humans have a limit, and AI is gradually beating us.

One of the other key reasons for this fast pace of improvement was that

researchers were openly sharing models trained on datasets like ImageNet. In the next section, we learn about model reuse in more detail.

researchers were openly sharing models trained on datasets like ImageNet. In the next section, we learn about model reuse in more detail.

Model Zoos

A model zoo is a place where organizations or individuals can publicly upload models that they have built for others to reuse and improve upon. These models can be trained using any framework (e.g., Keras, TensorFlow, MXNet), for any task (classification, detection, etc.), or trained on any dataset (e.g., ImageNet, Street View House Numbers (SVHN)).

The tradition of model zoos started with Caffe, one of the first deep learning frameworks, developed at the University of California, Berkeley. Training a deep learning model from scratch on a multimillion-image database requires weeks of training time and lots of GPU computational energy, making it a difficult task. The research community recognized this as a bottleneck, and the organizations that participated in the ImageNet competition open sourced their trained models on Caffe's website. Other frameworks soon followed suit.

When starting out on a new deep learning project, it's a good idea to first explore whether there's already a model that performs a similar task and was trained on a similar dataset.

The model zoo in Keras is a collection of various architectures trained using the Keras framework on the ImageNet dataset. We tabulate their details in Table 2-1

Table 2-1. Architectural details of select pretrained ImageNet models

Model	Size	Top-1 accuracy	Top-5 accuracy	Parameters	Depth
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.9	143,667,240	26
ResNet-50	98 MB	0.749	0.921	25,636,712	50
ResNet-101	171 MB	0.764	0.928	44,707,176	101
ResNet-152	232 MB	0.766	0.931	60,419,944	152
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
NASNetMobile	23 MB	0.744	0.919	5,326,716	—
NASNetLarge	343 MB	0.825	0.96	88,949,818	—
MobileNet	16 MB	0.704	0.895	4,253,864	88

MobileNetV2	14 MB	0.713	0.901	3,538,984	88

The column “Top-1 accuracy” indicates how many times the best guess was the correct answer, and the column “Top-5 accuracy” indicates how many times at least one out of five guesses were correct. The “Depth” of the network indicates how many layers are present in the network. The “Parameters” column indicates the size of the model; that is, how many individual weights the model has: the more parameters, the “heavier” the model is, and the slower it is to make predictions. In this book, we often use ResNet-50 (the most common architecture cited in research papers for high accuracy) and MobileNet (for a good balance between speed, size, and accuracy).

MobileNetV2	14 MB	0.713	0.901	3,538,984	88

The column “Top-1 accuracy” indicates how many times the best guess was the correct answer, and the column “Top-5 accuracy” indicates how many times at least one out of five guesses were correct. The “Depth” of the network indicates how many layers are present in the network. The “Parameters” column indicates the size of the model; that is, how many individual weights the model has: the more parameters, the “heavier” the model is, and the slower it is to make predictions. In this book, we often use ResNet-50 (the most common architecture cited in research papers for high accuracy) and MobileNet (for a good balance between speed, size, and accuracy).

Class Activation Maps

Image saliency, usually famous in UX research, is trying to answer the question "What part of the image are users paying attention to?" This is facilitated with the help of eye-tracking studies and represented in heatmaps. For example, big, bold fonts or people's faces usually get more attention than backgrounds. It's easy to guess how useful these heatmaps would be to designers and advertisers, who can then adapt their content to maximize users' attention. Taking inspiration from this human version of saliency, wouldn't it be great to learn which part of the image the neural network is paying attention to? That's precisely what we will be experimenting with.

In our experiment, we will be overlaying a *class activation map* (or colloquially a *heatmap*) on top of a video in order to understand what the network pays attention to. The heatmap tells us something like "In this picture, these pixels were responsible for the prediction of the class `dog` where "dog" was the category with the highest probability. The "hot" pixels are represented with warmer colors such as red, orange, and yellow, whereas the "cold" pixels are represented using blue. The "hotter" a pixel is, the higher the signal it provides toward the prediction. Figure 2-6 gives us a clearer picture. (If you're reading the print version, refer to the book's GitHub for the original color image.)

Figure 2-6. Original image of a dog and its generated heatmap

In the GitHub repository (see *http://PracticalDeepLearning.ai*), navigate to *code/chapter-2*. There, you'll find a handy Jupyter notebook, *2-class-activation-map-on-video.ipynb*, which describes the following steps:

First, we need to install `keras-vis` using `pip`:

```
$ pip install keras-vis --user
```

We then run the visualization script on a single image to generate the heatmap for it:

```
$ python visualization.py --process image --path ../sample-images/dog.jpg
```

We should see a newly created file called *dog-output.jpg* that shows a side-by-side view of the original image and its heatmap. As we can see from Figure 2-6, the right half of the image indicates the "areas of heat" along with the correct prediction of a "Cardigan" (i.e., Welsh Corgi).

Next, we want to visualize the heatmap for frames in a video. For that, we need `FFmpeg`, an open source multimedia framework. You can find the download binary as well as the installation instructions for your operating system at *https://www.ffmpeg.org*.

We use `ffmpeg` to split up a video into individual frames (at 25 frames per second) and then run our visualization script on each of those frames. We must first create a directory to store these frames and pass its name as part of the `ffmpeg` command:

```
$ mkdir kitchen
$ ffmpeg -i video/kitchen-input.mov -vf fps=25 kitchen/thumb%04d.jpg -hide_banner
```

We then run the visualization script with the path of the directory containing the frames from the previous step:

```
$ python visualization.py --process video --path kitchen/
```

We should see a newly created *kitchen-output* directory that contains all of the heatmaps for the frames from the input directory.

Finally, compile a video from those frames using `ffmpeg`:

```
$ ffmpeg -framerate 25 -i kitchen-output/result-%04d.jpg kitchen-output.mp4
```

Perfect! The result is the original video side by side with a copy of the heatmap overlaid on it. This is a useful tool, in particular, to discover whether the model has learned the correct features or if it picked up stray artifacts during its training.

Imagine generating heatmaps to analyze the strong points and shortfalls of our trained model or a pretrained model.

You should try this experiment out on your own by shooting a video with your smartphone camera and running the aforementioned scripts on the file. Don't forget to post your videos on Twitter, tagging @PracticalDLBook!

TIP

Heatmaps are a great way to visually detect bias in the data. The quality of a model's predictions depends heavily on the data on which it was trained. If the data is biased, that will reflect in the predictions. A great example of this is (although probably an urban legend) one in which the US Army wanted to use neural networks to detect enemy tanks camouflaged in trees.[1] The researchers who were building the model took photographs—50% containing camouflaged tanks and 50% with just trees. Model training yielded 100% accuracy. A cause for celebration? That sadly wasn't the case when the US Army tested it. The model had performed very poorly—no better than random guesses. Investigation revealed that photos with the tanks were taken on cloudy (overcast) days and those without the tanks on clear, sunny days. And the neural network model began looking for the sky instead of the tank. If the researchers had visualized the model using heatmaps, they would have caught that issue pretty early.

As we collect data, we must be vigilant at the outset of potential bias that can pollute our model's learning. For example, when collecting images to build a food classifier, we should verify that the other artifacts such as plates and utensils are not being learned as food. Otherwise, the presence of chopsticks might get our food classified as chow mein. Another term to define this is *co-occurrence*. Food very frequently co-occurs with cutlery. So watch out for these artifacts seeping into your classifier's training.

Finally, compile a video from those frames using `ffmpeg`:

```
$ ffmpeg -framerate 25 -i kitchen-output/result-%04d.jpg kitchen-output.mp4
```

Perfect! The result is the original video side by side with a copy of the heatmap overlaid on it. This is a useful tool, in particular, to discover whether the model has learned the correct features or if it picked up stray artifacts during its training.

Imagine generating heatmaps to analyze the strong points and shortfalls of our trained model or a pretrained model.

You should try this experiment out on your own by shooting a video with your smartphone camera and running the aforementioned scripts on the file. Don't forget to post your videos on Twitter, tagging @PracticalDLBook!

TIP

Heatmaps are a great way to visually detect bias in the data. The quality of a model's predictions depends heavily on the data on which it was trained. If the data is biased, that will reflect in the predictions. A great example of this is (although probably an urban legend) one in which the US Army wanted to use neural networks to detect enemy tanks camouflaged in trees.[1] The researchers who were building the model took photographs—50% containing camouflaged tanks and 50% with just trees. Model training yielded 100% accuracy. A cause for celebration? That sadly wasn't the case when the US Army tested it. The model had performed very poorly—no better than random guesses. Investigation revealed that photos with the tanks were taken on cloudy (overcast) days and those without the tanks on clear, sunny days. And the neural network model began looking for the sky instead of the tank. If the researchers had visualized the model using heatmaps, they would have caught that issue pretty early.

As we collect data, we must be vigilant at the outset of potential bias that can pollute our model's learning. For example, when collecting images to build a food classifier, we should verify that the other artifacts such as plates and utensils are not being learned as food. Otherwise, the presence of chopsticks might get our food classified as chow mein. Another term to define this is *co-occurrence*. Food very frequently co-occurs with cutlery. So watch out for these artifacts seeping into your classifier's training.

Summary

In this chapter, we got a glimpse of the deep learning universe using Keras. It's an easy-to-use yet powerful framework that we use in the next several chapters. We observed that there is often no need to collect millions of images and use powerful GPUs to train a custom model because we can use a pretrained model to predict the category of an image. By diving deeper into datasets like ImageNet, we learned the kinds of categories these pretrained models can predict. We also learned about finding these models in model zoos that exist for most frameworks.

In Chapter 3, we explore how we can tweak an existing pretrained model to make predictions on classes of input for which it was not originally intended. As with the current chapter, our approach is geared toward obtaining output without needing millions of images and lots of hardware resources to train a classifier.

---

1 "Artificial Intelligence as a Positive and Negative Factor in Global Risk" by Eliezer Yudkowsky in *Global Catastrophic Risks* (Oxford University Press).

Summary

In this chapter, we got a glimpse of the deep learning universe using Keras. It's an easy-to-use yet powerful framework that we use in the next several chapters. We observed that there is often no need to collect millions of images and use powerful GPUs to train a custom model because we can use a pretrained model to predict the category of an image. By diving deeper into datasets like ImageNet, we learned the kinds of categories these pretrained models can predict. We also learned about finding these models in model zoos that exist for most frameworks.

In Chapter 3, we explore how we can tweak an existing pretrained model to make predictions on classes of input for which it was not originally intended. As with the current chapter, our approach is geared toward obtaining output without needing millions of images and lots of hardware resources to train a classifier.

---

1 "Artificial Intelligence as a Positive and Negative Factor in Global Risk" by Eliezer Yudkowsky in *Global Catastrophic Risks* (Oxford University Press).

Chapter 3. Cats Versus Dogs: Transfer Learning in 30 Lines with Keras

Imagine that we want to learn how to play the melodica, a wind instrument in the form of a handheld keyboard. Without a musical background, and the melodica being our very first instrument, it might take us a few months to become proficient at playing it. In contrast, if we were already skilled at playing another instrument, such as the piano, it might just take a few days, given how similar the two instruments are. Taking the learnings from one task and fine tuning them on a similar task is something we often do in real life (as illustrated in Figure 3-1). The more similar the two tasks are, the easier it is to adapt the learning from one task to the other.

We can apply this phenomenon from real life to the world of deep learning. Starting a deep learning project can be relatively quick when using a pretrained model, which reuses the knowledge that it learned during its training, and adapt it to the task at hand. This process is known as *transfer learning*.

In this chapter, we use transfer learning to modify existing models by training our own classifier in minutes using Keras. By the end of this chapter, we will have several tools in our arsenal to create high-accuracy image classifiers for any task.

Figure 3-1. Transfer learning in real life

Figure 3-1. Transfer learning in real life

Adapting Pretrained Models to New Tasks

Before we discuss the process of transfer learning, let's quickly take a step back and review the primary reasons for the boom in deep learning:

- Availability of bigger and better-quality datasets like ImageNet
- Better compute available; i.e., faster and cheaper GPUs
- Better algorithms (model architecture, optimizer, and training procedure)
- Availability of pretrained models that have taken months to train but can be quickly reused

The last point is probably one of the biggest reasons for the widespread adoption of deep learning by the masses. If every training task took a month, not more than a handful of researchers with deep pockets would be working in this area. Thanks to transfer learning, the underappreciated hero of training models, we can now modify an existing model to suit our task in as little as a few minutes.

For example, we saw in Chapter 2 that the pretrained ResNet-50 model, which is trained on ImageNet, can predict feline and canine breeds, among thousands of other categories. So, if we just want to classify between the high-level "cat" and "dog" categories (and not the lower-level breeds), we can begin with the ResNet 50 model and quickly retrain this model to classify cats and dogs. All we need to do is show it a dataset with these two categories during training, which should take anywhere between a few minutes to a few hours. In comparison, if we had to train a cat versus dog model without a pretrained model, it could take several hours to days.

FROM THE CREATOR'S DESK

By Jeremy Howard, cofounder of fast.ai and former chief scientist at Kaggle

Hundreds of thousands of students have studied deep learning through fast.ai. Our goal is to get them up and running as quickly as possible, solving real problems quickly. So what's the first thing we teach? It's transfer learning!

Thousands of students have now shared their success stories on our forum (*http://forums.fast.ai*) describing how with as few as 30 images they have created 100% accurate image classifiers. We've also heard from students that have broken academic records in many domains and created commercially valuable models using this simple technique.

Five years ago, I created Enlitic, the first company to focus on deep learning for medicine. As an initial proof of concept, I decided to develop a lung tumor classifier from CT scans. You can probably guess what technique we used...yes, it was transfer learning! In our open source fast.ai library we make transfer learning trivially easy—it's just three lines of code, and the most important best practices are built in.

Thousands of students have now shared their success stories on our forum (*http://forums.fast.ai*) describing how with as few as 30 images they have created 100% accurate image classifiers. We've also heard from students that have broken academic records in many domains and created commercially valuable models using this simple technique.

Five years ago, I created Enlitic, the first company to focus on deep learning for medicine. As an initial proof of concept, I decided to develop a lung tumor classifier from CT scans. You can probably guess what technique we used...yes, it was transfer learning! In our open source fast.ai library we make transfer learning trivially easy—it's just three lines of code, and the most important best practices are built in.

A Shallow Dive into Convolutional Neural Networks

We have been using the term “model” to refer to the part of AI that makes our predictions. In deep learning for computer vision, that model is usually a special type of neural network called a CNN. Although we explore CNNs in greater detail later in the book, we look at them very briefly in relation to training them via transfer learning here.

In machine learning, we need to convert data into a set of discernible features and then add a classification algorithm to classify them. It’s the same with CNNs. They consist of two parts: convolutional layers and fully connected layers. The job of the convolutional layers is to take the large number of pixels of an image and convert them into a much smaller representation; that is, features. The fully connected layers convert these features into probabilities. A fully connected layer is really a neural network with hidden layers, as we saw in Chapter 1. In summary, the convolutional layers act as feature extractors, whereas the fully connected layers act as classifiers. Figure 3-2 shows a high-level overview of a CNN.

Figure 3-2. A high-level overview of a CNN

Imagine that we want to detect a human face. We might want to use a CNN to classify an image and determine whether it contains a face. Such a CNN would

be composed of several layers connected one after another. These layers represent mathematical operations. The output of one layer is the input to the next. The first (or the lowermost) layer is the input layer, where the input image is fed. The last (or the topmost) layer is the output layer, which gives the predictions.

The way it works is the image is fed into the CNN and passes through a series of layers, with each performing a mathematical operation and passing the result to the subsequent layer. The resulting output is a list of object classes and their probabilities. For example, categories like ball—65%, grass—20%, and so on. If the output for an image contains a “face” class with a 70% probability, we conclude that there is a 70% likelihood that the image contains a human face.

NOTE

An intuitive (and overly simplified) way to look at CNNs is to see them as a series of filters. As the word filter implies, each layer acts as a sieve of information, letting something “pass through” only if it recognizes it. (If you have heard of high-pass and low-pass filters in electronics, this might seem familiar.) We say that the layer was “activated” for that information. Each layer is activated for visual patterns resembling parts of cats, dogs, cars, and so forth. If a layer does not recognize information (due to what it learned while training), its output is close to zero. CNNs are the “bouncers” of the deep learning world!

In the facial detection example, lower-level layers (Figure 3-3, a; layers that are closer to the input image) are “activated” for simpler shapes; for example, edges and curves. Because these layers activate only for basic shapes, they can be easily reused for a different purpose than face recognition such as detecting a car (every image is composed of edges and curves, after all). Middle-level layers (Figure 3-3 b) are activated for more complex shapes such as eyes, noses, and lips. These layers are not nearly as reusable as the lower-level layers. They might not be as useful for detecting a car, but might still be useful for detecting animals. And higher-level layers (Figure 3-3 c) are activated for even more complex shapes; for example, most of the human face. These layers tend to be more task-specific and thus the least reusable across other image classification problems.

Figure 3-3. (a) Lower-level activations, followed by (b) midlevel activations and (c) upper-layer activation (image source: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations, Lee et al., ICML 2009)

The complexity and power of what a layer can recognize increases as we approach the final layers. Conversely, the reusability of a layer decreases as we get closer to the output. This will become apparent very soon when we look at what these layers learn.

Figure 3-3. (a) Lower-level activations, followed by (b) midlevel activations and (c) upper-layer activations (image source: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations, Lee et al., ICML 2009)

The complexity and power of what a layer can recognize increases as we approach the final layers. Conversely, the reusability of a layer decreases as we get closer to the output. This will become apparent very soon when we look at what these layers learn.

Transfer Learning

If we want to transfer knowledge from one model to another, we want to reuse more of the *generic* layers (closer to the input) and fewer of the *task-specific* layers (closer to the output). In other words, we want to remove the last few layers (typically the fully connected layers) so that we can utilize the more generic ones, and add layers that are geared toward our specific classification task. Once training begins, the generic layers (which form the majority of our new model) are kept frozen (i.e., they are unmodifiable), whereas the newly added task-specific layers are allowed to be modified. This is how transfer learning helps quickly train new models. Figure 3-4 illustrates this process for a pretrained model trained for task X adapted to task Y.

Figure 3-4. An overview of transfer learning

Transfer Learning

If we want to transfer knowledge from one model to another, we want to reuse more of the *generic* layers (closer to the input) and fewer of the *task-specific* layers (closer to the output). In other words, we want to remove the last few layers (typically the fully connected layers) so that we can utilize the more generic ones, and add layers that are geared toward our specific classification task. Once training begins, the generic layers (which form the majority of our new model) are kept frozen (i.e., they are unmodifiable), whereas the newly added task-specific layers are allowed to be modified. This is how transfer learning helps quickly train new models. Figure 3-4 illustrates this process for a pretrained model trained for task X adapted to task Y.

Figure 3-4. An overview of transfer learning

Fine Tuning

Basic transfer learning gets us only so far. We usually add only two to three fully connected layers after the generic layers to make the new classifier model. If we want higher accuracy, we must allow more layers to be trained. This means unfreezing some of the layers that would have otherwise been frozen in transfer learning. This is known as *fine tuning*. Figure 3-5 shows an example where some convolutional layers near the head/top are unfrozen and trained for the task at hand.

Figure 3-5. Fine tuning a convolutional neural network

It's obvious that compared to basic transfer learning, more layers are tweaked to our dataset during fine tuning. Because a higher number of layers have adapted to our task compared to transfer learning, we can achieve greater accuracy for our task. The decision on how many layers to fine tune is dependent on the amount of data at hand as well as the similarity of the target task to the original dataset on which the pretrained model was trained.

We often hear data scientists saying, "I fine tuned the model," which means that they took a pretrained model, removed task-specific layers and added new ones, froze the lower layers, and trained the upper part of the network on the new

dataset they had.

> **NOTE**
>
> In daily lingo, transfer learning and fine tuning are used interchangeably. When spoken, transfer learning is used more as a general concept, whereas fine tuning is referred to as its implementation.

dataset they had.

NOTE

In daily lingo, transfer learning and fine tuning are used interchangeably. When spoken, transfer learning is used more as a general concept, whereas fine tuning is referred to as its implementation.

How Much to Fine Tune

How many layers of a CNN should we fine tune? This can be guided by the following two factors:

How much data do we have?

If we have a couple hundred labeled images, it would be difficult to train and test a freshly defined model from scratch (i.e., define a model architecture with random seed weights) because we need a lot more data. The danger of training with such a small amount of data is that these powerful networks might potentially memorize it, leading to undesirable overfitting (which we explore later in the chapter). Instead, we will borrow a pretrained network and fine tune the last few layers. But if we had a million labeled images, it would be feasible to fine tune all layers of the network and, if necessary, train from scratch. So, the amount of task-specific data dictates whether we can fine tune, and how much.

How similar is the data?

If the task-specific data is similar to the data used for the pretrained network, we can fine tune the last few layers. But if our task is identifying different bones in an X-ray image and we want to start out from an ImageNet trained network, the high dissimilarity between regular ImageNet images and X-ray images would require nearly all layers to be trained.

To summarize, Table 3-1 offers an easy-to-follow cheat sheet.

Table 3-1. Cheatsheet for when and how to fine tune

	High similarity among datasets	Low similarity among datasets
Large amount of training data	Fine tune all layers	Train from scratch, or fine tune all layers
Small amount of training data	Fine tune last few layers	Tough luck! Train on a smaller network with heavy data augmentation or somehow get more data

Enough theory, let’s see it in action.

Building a Custom Classifier in Keras with Transfer Learning

As promised, it's time to build our state-of-the-art classifier in 30 lines or less. At a high level, we will use the following steps:

1. Organize the data. Download labeled images of cats and dogs and then divide the images into training and validation folders.
2. Build the data pipeline. Define a pipeline for reading data, including preprocessing the images (e.g., resizing) and grouping multiple images together into batches.
3. Augment the data. In the absence of a ton of training images, make small changes (augmentation) like rotation, zooming, and so on to increase variation in training data.
4. Define the model. Take a pretrained model, remove the last few task-specific layers, and append a new classifier layer. Freeze the weights of original layers (i.e., make them unmodifiable). Select an optimizer algorithm and a metric to track (like accuracy).
5. Train and test. Train for a few iterations until our validation accuracy is high. Save the model to eventually load as part of any application for predictions.

This will all make sense pretty soon. Let's explore this process in detail.

SOLVING THE WORLD'S MOST PRESSING COMPUTER-VISION PROBLEM

In early 2014, Microsoft Research was figuring out how to solve the world's most pressing problem at the time: "Differentiating cats and dogs." (Where else would we have gotten the idea for this chapter?) Keep in mind that it was a much more difficult computer-vision problem back then. To facilitate this effort, Microsoft Research released the Asirra (Animal Species Image Recognition for Restricting Access) dataset. The motivation behind the

Asirra dataset was to develop a sufficiently challenging CAPTCHA system. More than three million images, labeled by animal shelters throughout the United States, were provided by *Petfinder.com* to Microsoft Research. When this problem was initially introduced, the highest possible accuracy attained was around 80%. By using deep learning, in just a few weeks, it went to 98%! This (now relatively easy) task shows the power of deep learning.

Asirra dataset was to develop a sufficiently challenging CAPTCHA system. More than three million images, labeled by animal shelters throughout the United States, were provided by *Petfinder.com* to Microsoft Research. When this problem was initially introduced, the highest possible accuracy attained was around 80%. By using deep learning, in just a few weeks, it went to 98%! This (now relatively easy) task shows the power of deep learning.

Organize the Data

It's essential to understand the distinction between train, validation, and test data Let's look at a real-world analogy of a student preparing for standardized exams (e.g., SAT in the US, the Gaokao in China, JEE in India, CSAT in Korea, etc.). The in-class instruction and homework assignments are analogous to the training process. The quizzes, midterms, and other tests in school are the equivalent to the validation—the student is able to take them frequently, assess performance, and make improvements in their study plan. They're ultimately optimizing for their performance in the final standardized exam for which they get only one chance. The final exam is equivalent to the test set—the student does not get an opportunity to improve here (ignoring the ability to retake the test). This is their one shot at showing what they have learned.

Similarly, our aim is to give the best predictions in the real world. To enable this, we divide our data into three parts: train, validation, and test. A typical distribution would be 80% for train, 10% for validation, and 10% for test. Note that we randomly divide our data into these three sets in order to ensure the least amount of *bias* that might creep in unknowingly. The final accuracy of the mode is determined by the accuracy on the *test set*, much like the student's score is determined only on their performance on the standardized exam.

The model learns from the training data and uses the validation set to evaluate its performance. Machine learning practitioners take this performance as feedback to find opportunities to improve their models on a continuous basis, similar to how students improve their preparation with the help of quizzes. There are several knobs that we can tune to improve performance; for example, the number of layers to train.

In many research competitions (including *Kaggle.com*), contestants receive a tes set that is separate from the data they can use for building the model. This ensures uniformity across the competition when it comes to reporting accuracy. It is up to the contestants to divide the available data into training and validation sets. Similarly, during our experiments in this book, we will continue to divide data in these two sets, keeping in mind that a test dataset is still essential to report real-world numbers.

So why even use a validation set? Data is sometimes difficult to obtain, so why not use all the available samples for training, and then report accuracy on them? Sure, when the model begins to learn, it will gradually give higher accuracy predictions on the training dataset (called training accuracy). But because they are so powerful, deep neural networks can potentially memorize the training data, even resulting in 100% accuracy on the training data sometimes. However, its real-world performance will be quite poor. It's like if the student knew the questions that would be on the exam before taking it. This is why a validation set, not used to train the model, gives a realistic assessment of the model performance. Even though we might assign 10-15% of the data as a validation set, it will go a long way in guiding us on how good our model really is.

For the training process, we need to store our dataset in the proper folder structure. We'll divide the images into two sets: training and validation. For an image file, Keras will automatically assign the name of the *class* (category) based on its parent folder name. Figure 3-6 depicts the ideal structure to recreate

Figure 3-6. Example directory structure of the training and validation data for different classes

The following sequence of commands can help download the data and achieve this directory structure:

```
$ wget https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/
download/train.zip
$ unzip train.zip
$ mv train data
$ cd data
$ mkdir train val
$ mkdir train/cat train/dog
$ mkdir val/cat val/dog
```

The 25,000 files within the data folder are prefixed with "cat" and "dog." Now, move the files into their respective directories. To keep our initial experiment

short, we pick 250 random files per class and place them in training and validation folders. We can increase/decrease this number anytime, to experiment with a trade-off between accuracy and speed:

```
$ ls | grep cat | sort -R | head -250 | xargs -I {} mv {} train/cat/
$ ls | grep dog | sort -R | head -250 | xargs -I {} mv {} train/dog/
$ ls | grep cat | sort -R | head -250 | xargs -I {} mv {} val/cat/
$ ls | grep dog | sort -R | head -250 | xargs -I {} mv {} val/dog/
```

.

short, we pick 250 random files per class and place them in training and validation folders. We can increase/decrease this number anytime, to experiment with a trade-off between accuracy and speed:

```
$ ls | grep cat | sort -R | head -250 | xargs -I {} mv {} train/cat/
$ ls | grep dog | sort -R | head -250 | xargs -I {} mv {} train/dog/
$ ls | grep cat | sort -R | head -250 | xargs -I {} mv {} val/cat/
$ ls | grep dog | sort -R | head -250 | xargs -I {} mv {} val/dog/
```

Build the Data Pipeline

To start off with our Python program, we begin by importing the necessary packages:

```
import tensorflow as tf
from tf.keras.preprocessing.image import ImageDataGenerator
from tf.keras.models import Model
from tf.keras.layers import Input, Flatten, Dense, Dropout,
GlobalAveragePooling2D
from tf.keras.applications.mobilenet import MobileNet, preprocess_input
import math
```

Place the following lines of configuration right after the import statements, which we can modify based on our dataset:

```
TRAIN_DATA_DIR = 'data/train_data/'
VALIDATION_DATA_DIR = 'data/val_data/'
TRAIN_SAMPLES = 500
VALIDATION_SAMPLES = 500
NUM_CLASSES = 2
IMG_WIDTH, IMG_HEIGHT = 224, 224
BATCH_SIZE = 64
```

Build the Data Pipeline

To start off with our Python program, we begin by importing the necessary packages:

```
import tensorflow as tf
from tf.keras.preprocessing.image import ImageDataGenerator
from tf.keras.models import Model
from tf.keras.layers import Input, Flatten, Dense, Dropout,
GlobalAveragePooling2D
from tf.keras.applications.mobilenet import MobileNet, preprocess_input
import math
```

Place the following lines of configuration right after the import statements, which we can modify based on our dataset:

```
TRAIN_DATA_DIR = 'data/train_data/'
VALIDATION_DATA_DIR = 'data/val_data/'
TRAIN_SAMPLES = 500
VALIDATION_SAMPLES = 500
NUM_CLASSES = 2
IMG_WIDTH, IMG_HEIGHT = 224, 224
BATCH_SIZE = 64
```

Number of Classes

With two classes to distinguish between, we can treat this problem as one of the following:

- A binary classification task
- A multiclass classification task

Binary classification

As a binary classification task, it's important to note that "cat versus dog is really "cat versus not cat." A dog would be classified as a "not cat" much like a desk or a ball would. For a given image, the model will give a single probability value corresponding to the "cat" class—hence the probability of "not cat" is 1 - *P(cat)*. If the probability is higher than 0.5, we predict it as "cat"; otherwise, "not cat." To keep things simple, we assume that it's guaranteed that the test set would contain only images of either cats or dogs. Because "cat versus not cat" is a binary classification task, we set the number of classes to 1; that is, "cat." Anything that cannot be classified as "cat" will be classified as "not cat."

TIP

Keras processes the input data in the alphabetical order of the folder names. Because "cat" comes before "dog" alphabetically, our first class for prediction is "cat." For a multiclass task, we can apply the same concept and infer each class identifier (index) based on the folder sort order. Note that the class index starts at 0 for the first class.

Multiclass classification

In a hypothetical world that had only cats and dogs and nothing else, a "not cat" would always be a dog. So the label "not cat" could simply be replaced with the label "dog." However, in the real world, we have more than two types of objects. As explained before, a ball or a sofa would also be classified as "dog," which would be incorrect. Hence, for a real-world scenario, treating this as a multiclass classification task instead of a binary classification task is far more useful. As a multiclass classification task, we predict separate probability values for each

class, and the highest one is our winner. In the case of “cat versus dog,” we set the number of classes to two. To keep our code reusable for future tasks, we will treat this as a multiclassification task.

class, and the highest one is our winner. In the case of “cat versus dog,” we set the number of classes to two. To keep our code reusable for future tasks, we will treat this as a multiclassification task.

Batch Size

At a high level, the training process includes the following steps:

1. Make predictions on images (*forward pass*).
2. Determine which predictions were incorrect and propagate back the difference between the prediction and the true value (*backpropagation*).
3. Rinse and repeat until the predictions become sufficiently accurate.

It's quite likely that the initial iteration would have close to 0% accuracy. Repeating the process several times, however, can yield a highly accurate model (>90%).

The batch size defines how many images are seen by the model at a time. It's important that each batch has a good variety of images from different classes in order to prevent large fluctuations in the accuracy metric between iterations. A sufficiently large batch size would be necessary for that. However, it's important not to set the batch size too large; a batch that is too large might not fit in GPU memory, resulting in an "out of memory" crash. Usually, batch sizes are set as powers of 2. A good number to start with is 64 for most problems, and we can play with the number by increasing or decreasing it.

Batch Size

At a high level, the training process includes the following steps:

1. Make predictions on images (*forward pass*).
2. Determine which predictions were incorrect and propagate back the difference between the prediction and the true value (*backpropagation*).
3. Rinse and repeat until the predictions become sufficiently accurate.

It's quite likely that the initial iteration would have close to 0% accuracy. Repeating the process several times, however, can yield a highly accurate model (>90%).

The batch size defines how many images are seen by the model at a time. It's important that each batch has a good variety of images from different classes in order to prevent large fluctuations in the accuracy metric between iterations. A sufficiently large batch size would be necessary for that. However, it's important not to set the batch size too large; a batch that is too large might not fit in GPU memory, resulting in an "out of memory" crash. Usually, batch sizes are set as powers of 2. A good number to start with is 64 for most problems, and we can play with the number by increasing or decreasing it.

Data Augmentation

Usually, when we hear deep learning, we associate it with millions of images. So, 500 images like what we have might be a low number for real-world training. While these deep neural networks are powerful, a little too powerful for small quantities of data, the danger of a limited set of training images is that the neural network might memorize our training data, and show great prediction performance on the training set, but bad accuracy on the validation set. In other words, the model has overtrained and does not generalize on previously unseen images. And we definitely don't want that.

TIP

Often, when we attempt to train a neural network on a small amount of data, the result is a model that performs extremely well on the training data itself but makes rather poor predictions on data that it has not seen before. Such a model would be described as an *overfitted* model and the problem itself is known as *overfitting*.

Figure 3-7 illustrates this phenomenon for a distribution of points close to a sine curve (with little noise). The dots represent the training data visible to our network, and the crosses represent the testing data that was not seen during training. On one extreme (underfitting), an unsophisticated model, such as a linear predictor, will not be able to represent the underlying distribution well and a high error rate on both the training data and the test data will result. On the other extreme (overfitting), a powerful model (such as a deep neural network) might have the capacity to memorize the training data, which would result in a really low error on the training data, but still a high error on the testing data. What we want is the happy middle where the training error and the testing error are both modestly low, which ideally ensures that our model will perform just as well in the real world as it does during training.

Figure 3-7. Underfitting, overfitting, and ideal fitting for points close to a sine curve

With great power comes great responsibility. It's our responsibility to ensure that our powerful deep neural network does not overfit on our data. Overfitting is common when we have little training data. We can reduce this likelihood in a few different ways:

- Somehow get more data
- Heavily augment existing data
- Fine tune fewer layers

There are often situations for which there's not enough data available. Perhaps we're working on a niche problem and data is difficult to come by. But there are a few ways that we can artificially augment our dataset for classification:

Rotation

In our example, we might want to rotate the 500 images randomly by 20 degrees in either direction, yielding up to 20,000 possible unique images.

Random Shift

Shift the images slightly to the left, or to the right.

Zoom

Zoom in and out slightly of the image.

By combining rotation, shifting, and zooming, the program can generate an almost infinite number of unique images. This important step is called *data*

augmentation. Data augmentation is useful not only for adding more data, but also for training more robust models for real-world scenarios. For example, not all images have the cat properly centered in the middle or at a perfect 0-degree angle. Keras provides the `ImageDataGenerator` function that augments the data while it is being loaded from the directory. To illustrate what data augmentations of images look like, Figure 3-8 showcases example augmentations generated by the imgaug library for a sample image. (Note that we will not be using imgaug for our actual training.)

Figure 3-8. Possible image augmentations generated from a single image

Colored images usually have three channels: red, green, and blue. Each channel has an intensity value ranging from 0 to 255. To normalize it (i.e., scale down the value to between 0 and 1), we use the `preprocess_input` function (which, among other things, divides each pixel by 255):

```
train_datagen = ImageDataGenerator(preprocessing_function=preprocess_input,
                                   rotation_range=20,
                                   width_shift_range=0.2,
```

```
                                    height_shift_range=0.2,
                                    zoom_range=0.2)
val_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
```

TIP

Sometimes knowing the label of a training image can be useful in determining appropriate ways of augmenting it. For example, when training a digit recognizer, you might be okay with augmentation by flipping vertically for an image of the digit “8,” but not for “6” and “9.”

Unlike our training set, we don’t want to augment our validation set. The reason is that with dynamic augmentation, the validation set would keep changing in each iteration, and the resulting accuracy metric would be inconsistent and difficult to compare across other iterations.

It’s time to load the data from its directories. Training one image at a time can be pretty inefficient, so we can batch them into groups. To introduce more randomness during the training process, we’ll keep shuffling the images in each batch. To bring reproducibility during multiple runs of the same program, we’ll give the random number generator a seed value:

```
train_generator = train_datagen.flow_from_directory(
                        TRAIN_DATA_DIR,
                        target_size=(IMG_WIDTH, IMG_HEIGHT),
                        batch_size=BATCH_SIZE,
                        shuffle=True,
                        seed=12345,
                        class_mode='categorical')
validation_generator = val_datagen.flow_from_directory(
                        VALIDATION_DATA_DIR,
                        target_size=(IMG_WIDTH, IMG_HEIGHT),
                        batch_size=BATCH_SIZE,
                        shuffle=False,
                        class_mode='categorical')
```

```
                                    height_shift_range=0.2,
                                    zoom_range=0.2)
val_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
```

TIP

Sometimes knowing the label of a training image can be useful in determining appropriate ways of augmenting it. For example, when training a digit recognizer, you might be okay with augmentation by flipping vertically for an image of the digit “8,” but not for “6” and “9.”

Unlike our training set, we don’t want to augment our validation set. The reason is that with dynamic augmentation, the validation set would keep changing in each iteration, and the resulting accuracy metric would be inconsistent and difficult to compare across other iterations.

It’s time to load the data from its directories. Training one image at a time can be pretty inefficient, so we can batch them into groups. To introduce more randomness during the training process, we’ll keep shuffling the images in each batch. To bring reproducibility during multiple runs of the same program, we’ll give the random number generator a seed value:

```
train_generator = train_datagen.flow_from_directory(
                        TRAIN_DATA_DIR,
                        target_size=(IMG_WIDTH, IMG_HEIGHT),
                        batch_size=BATCH_SIZE,
                        shuffle=True,
                        seed=12345,
                        class_mode='categorical')
validation_generator = val_datagen.flow_from_directory(
                        VALIDATION_DATA_DIR,
                        target_size=(IMG_WIDTH, IMG_HEIGHT),
                        batch_size=BATCH_SIZE,
                        shuffle=False,
                        class_mode='categorical')
```

Model Definition

Now that the data is taken care of, we come to the most crucial component of our training process: the model. In the code that follows, we reuse a CNN previously trained on the ImageNet dataset (MobileNet in our case), throw away the last few layers, called fully connected layers (i.e., ImageNet-specific classifier layers), and replace them with our own classifier suited to the task at hand.

For transfer learning, we "freeze" the weights of the original model; that is, set those layers as unmodifiable, so only the layers of the new classifier (that we'll add) can be modified. We use MobileNet here to keep things fast, but this method will work just as well for any neural network. The following lines include a few terms such as `Dense`, `Dropout`, and so on. Although we won't explore them in this chapter, you can find explanations in Appendix A.

```
def model_maker():
    base_model = MobileNet(include_top=False, input_shape =
(IMG_WIDTH,IMG_HEIGHT,3))
    for layer in base_model.layers[:]:
        layer.trainable = False # Freeze the layers
    input = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3))
    custom_model = base_model(input)
    custom_model = GlobalAveragePooling2D()(custom_model)
    custom_model = Dense(64, activation='relu')(custom_model)
    custom_model = Dropout(0.5)(custom_model)
    predictions = Dense(NUM_CLASSES, activation='softmax')(custom_model)
    return Model(inputs=input, outputs=predictions)
```

Model Definition

Now that the data is taken care of, we come to the most crucial component of our training process: the model. In the code that follows, we reuse a CNN previously trained on the ImageNet dataset (MobileNet in our case), throw away the last few layers, called fully connected layers (i.e., ImageNet-specific classifier layers), and replace them with our own classifier suited to the task at hand.

For transfer learning, we "freeze" the weights of the original model; that is, set those layers as unmodifiable, so only the layers of the new classifier (that we'll add) can be modified. We use MobileNet here to keep things fast, but this method will work just as well for any neural network. The following lines include a few terms such as `Dense`, `Dropout`, and so on. Although we won't explore them in this chapter, you can find explanations in Appendix A.

```
def model_maker():
    base_model = MobileNet(include_top=False, input_shape =
(IMG_WIDTH,IMG_HEIGHT,3))
    for layer in base_model.layers[:]:
        layer.trainable = False # Freeze the layers
    input = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3))
    custom_model = base_model(input)
    custom_model = GlobalAveragePooling2D()(custom_model)
    custom_model = Dense(64, activation='relu')(custom_model)
    custom_model = Dropout(0.5)(custom_model)
    predictions = Dense(NUM_CLASSES, activation='softmax')(custom_model)
    return Model(inputs=input, outputs=predictions)
```

Train the Model

Train the Model

Set Training Parameters

With both the data and model ready, all we have left to do is train the model. This is also known as *fitting the model to the data*. For training a model, we neec to select and modify a few different training parameters.

Loss function

The `loss` function is the penalty we impose on the model for incorrect predictions during the training process. It is the value of this function that we seek to *minimize*. For example, in a task to predict house prices, the `loss` function could be the root-mean-square error.

Optimizer

This is an algorithm that helps minimize the `loss` function. We use Adam, one of the fastest optimizers out there.

Learning rate

Learning is incremental. The learning rate tells the optimizer how big of a step to take toward the solution; in other words, where the loss is minimum. Take too big of a step, and we end up wildly swinging and overshooting our target. Take too small a step, and it can take a really long time before eventually arriving at the target loss value. It is important to set an optimal learning rate to ensure that we reach our learning goal in a reasonable amount of time. In our example, we set the learning rate at 0.001.

Metric

Choose a metric to judge the performance of the trained model. Accuracy is a good explainable metric, especially when the classes are not imbalanced (i.e., roughly equal amounts of data for each class). Note that this metric is not related to the `loss` function and is mainly used for reporting and not as feedback for the model.

In the following piece of code, we create the custom model using the `model_maker` function that we wrote earlier. We use the parameters described here to customize this model further for our task of cats versus dogs:

```
model = model_maker()
model.compile(loss='categorical_crossentropy',
              optimizer= tf.train.Adam(lr=0.001),
              metrics=['acc'])
num_steps = math.ceil(float(TRAIN_SAMPLES)/BATCH_SIZE)
model.fit_generator(train_generator,
                    steps_per_epoch = num_steps,
                    epochs=10,
                    validation_data = validation_generator,
                    validation_steps = num_steps)
```

NOTE

You might have noticed the term epoch in the preceding code. One epoch represents a full training step where the network has gone over the entire dataset. One epoch may consist of several minibatches.

```
model = model_maker()
model.compile(loss='categorical_crossentropy',
              optimizer= tf.train.Adam(lr=0.001),
              metrics=['acc'])
num_steps = math.ceil(float(TRAIN_SAMPLES)/BATCH_SIZE)
model.fit_generator(train_generator,
                    steps_per_epoch = num_steps,
                    epochs=10,
                    validation_data = validation_generator,
                    validation_steps = num_steps)
```

NOTE

You might have noticed the term epoch in the preceding code. One epoch represents a full training step where the network has gone over the entire dataset. One epoch may consist of several minibatches.

Start Training

Run this program and let the magic begin. If you don't have a GPU, brew a cup of coffee while you wait—it might take 5 to 10 minutes. Or why wait, when you can run the notebooks of this chapter (posted on GitHub) on Colab with a GPU runtime for free?

When complete, notice that there are four statistics: `loss` and `acc` on both the training and validation data. We are rooting for `val_acc:`

```
> Epoch 1/100 7/7 [====] - 5s -
loss: 0.6888 - acc: 0.6756 - val_loss: 0.2786 - val_acc: 0.9018
> Epoch 2/100 7/7 [====] - 5s -
loss: 0.2915 - acc: 0.9019 - val_loss: 0.2022 - val_acc: 0.9220
> Epoch 3/100 7/7 [====] - 4s -
loss: 0.1851 - acc: 0.9158 - val_loss: 0.1356 - val_acc: 0.9427
> Epoch 4/100 7/7 [====] - 4s -
loss: 0.1509 - acc: 0.9341 - val_loss: 0.1451 - val_acc: 0.9404
> Epoch 5/100 7/7 [====] - 4s -
loss: 0.1455 - acc: 0.9464 - val_loss: 0.1637 - val_acc: 0.9381
> Epoch 6/100 7/7 [====] - 4s -
loss: 0.1366 - acc: 0.9431 - val_loss: 0.2319 - val_acc: 0.9151
> Epoch 7/100 7/7 [====] - 4s -
loss: 0.0983 - acc: 0.9606 - val_loss: 0.1420 - val_acc: 0.9495
> Epoch 8/100 7/7 [====] - 4s -
loss: 0.0841 - acc: 0.9731 - val_loss: 0.1423 - val_acc: 0.9518
> Epoch 9/100 7/7 [====] - 4s -
loss: 0.0714 - acc: 0.9839 - val_loss: 0.1564 - val_acc: 0.9509
> Epoch 10/100 7/7 [====] - 5s -
loss: 0.0848 - acc: 0.9677 - val_loss: 0.0882 - val_acc: 0.9702
```

All it took was 5 seconds in the very first epoch to reach 90% accuracy on the validation set, with just 500 training images. Not bad! And by the 10th step, we observe about 97% *validation accuracy*. That's the power of transfer learning.

Let us take a moment to appreciate what happened here. With just 500 images, we were able to reach a high level of accuracy in a matter of a few seconds and with very little code. In contrast, if we did not have a model previously trained on ImageNet, getting an accurate model might have needed training time anywhere between a couple of hours to a few days, and tons more data.

That's all the code we need to train a state-of-the-art classifier on any problem.

Place data into folders with the name of the class, and change the corresponding values in the configuration variables. In case our task has more than two classes, we should use `categorical_crossentropy` as the `loss` function and replace the `activation` function in the last layer with `softmax`. Table 3-2 illustrates this.

Table 3-2. Deciding the loss and activation type based on the task

Classification type	Class mode	Loss	Activation on the last layer
1 or 2 classes	binary	binary_crossentropy	sigmoid
Multiclass, single label	categorical	categorical_crossentropy	softmax
Multiclass, multilabel	categorical	binary_crossentropy	sigmoid

Before we forget, save the model that you just trained so that we can use it later:

```
model.save('model.h5')
```

Place data into folders with the name of the class, and change the corresponding values in the configuration variables. In case our task has more than two classes, we should use `categorical_crossentropy` as the `loss` function and replace the `activation` function in the last layer with `softmax`. Table 3-2 illustrates this.

Table 3-2. Deciding the loss and activation type based on the task

Classification type	Class mode	Loss	Activation on the last layer
1 or 2 classes	binary	binary_crossentropy	sigmoid
Multiclass, single label	categorical	categorical_crossentropy	softmax
Multiclass, multilabel	categorical	binary_crossentropy	sigmoid

Before we forget, save the model that you just trained so that we can use it later:

```
model.save('model.h5')
```

Test the Model

Now that we have a trained model, we might eventually want to use it later for our application. We can now load this model anytime and classify an image. `load_model`, as its name suggests, loads the model:

```
from tf.keras.models import load_model
model = load_model('model.h5')
```

Now let's try loading our original sample images and see what results we get:

```
img_path = '../../sample_images/dog.jpg'
img = image.load_img(img_path, target_size=(224,224))
img_array = image.img_to_array(img)
expanded_img_array = np.expand_dims(img_array, axis=0)
preprocessed_img = preprocess_input(expanded_img_array) # Preprocess the image
prediction = model.predict(preprocessed_img)
print(prediction)
print(validation_generator.class_indices)
[[0.9967706]]
{'dog': 1, 'cat': 0}
```

Printing the value of the probability, we see that it is 0.996. This is the probability of the given image belonging to the class "1," which is a dog. Because the probability is greater than 0.5, the image is predicted as a dog.

That's all that we need to train our own classifiers. Throughout this book, you can expect to reuse this code for training with minimal modifications. You can also reuse this code in your own projects. Play with the number of epochs and images, and observe how it affects the accuracy. Also, we should play with any other data we can find online. It doesn't get easier than that!

Test the Model

Now that we have a trained model, we might eventually want to use it later for our application. We can now load this model anytime and classify an image. `load_model`, as its name suggests, loads the model:

```
from tf.keras.models import load_model
model = load_model('model.h5')
```

Now let's try loading our original sample images and see what results we get:

```
img_path = '../../sample_images/dog.jpg'
img = image.load_img(img_path, target_size=(224,224))
img_array = image.img_to_array(img)
expanded_img_array = np.expand_dims(img_array, axis=0)
preprocessed_img = preprocess_input(expanded_img_array) # Preprocess the image
prediction = model.predict(preprocessed_img)
print(prediction)
print(validation_generator.class_indices)
[[0.9967706]]
{'dog': 1, 'cat': 0}
```

Printing the value of the probability, we see that it is 0.996. This is the probability of the given image belonging to the class "1," which is a dog. Because the probability is greater than 0.5, the image is predicted as a dog.

That's all that we need to train our own classifiers. Throughout this book, you can expect to reuse this code for training with minimal modifications. You can also reuse this code in your own projects. Play with the number of epochs and images, and observe how it affects the accuracy. Also, we should play with any other data we can find online. It doesn't get easier than that!

Analyzing the Results

With our trained model, we can analyze how it's performing on the validation dataset. Beyond the more straightforward accuracy metrics, looking at the actual images of mispredictions should give an intuition as to whether the example was truly challenging or whether our model is not sophisticated enough.

There are three questions that we want to answer for each category (cat, dog):

- Which images are we most confident about being a cat/dog?
- Which images are we least confident about being a cat/dog?
- Which images have incorrect predictions in spite of being highly confident?

Before we get to that, let's get predictions over the entire validation dataset. First, we set the pipeline configuration correctly:

```
# VARIABLES
IMG_WIDTH, IMG_HEIGHT = 224, 224
VALIDATION_DATA_DIR = 'data/val_data/'
VALIDATION_BATCH_SIZE = 64

# DATA GENERATORS
validation_datagen = ImageDataGenerator(
        preprocessing_function=preprocess_input)
validation_generator = validation_datagen.flow_from_directory(
        VALIDATION_DATA_DIR,
        target_size=(IMG_WIDTH, IMG_HEIGHT),
        batch_size=VALIDATION_BATCH_SIZE,
        shuffle=False,
        class_mode='categorical')
ground_truth = validation_generator.classes
```

Then, we get the predictions:

```
predictions = model.predict_generator(validation_generator)
```

To make our analysis easier, we make a dictionary storing the image index to the prediction and ground truth (the expected prediction) for each image:

```
# prediction_table is a dict with index, prediction, ground truth
prediction_table = {}
for index, val in enumerate(predictions):
    # get argmax index
    index_of_highest_probability = np.argmax(val)
    value_of_highest_probability = val[index_of_highest_probability]
    prediction_table[index] = [value_of_highest_probability,
index_of_highest_probability, ground_truth[index]]
assert len(predictions) == len(ground_truth) == len(prediction_table)
```

For the next two code blocks, we provide boilerplate code, which we reuse regularly throughout the book.

The following is the signature of the helper function we'll use to find the images with the highest/lowest probability value for a given category. Additionally, we will be using another helper function, - `display`(), to output the images as a grid on-screen:

```
def display(sorted_indices, message):
    similar_image_paths = []
    distances = []
    for name, value in sorted_indices:
        [probability, predicted_index, gt] = value
        similar_image_paths.append(VALIDATION_DATA_DIR + fnames[name])
        distances.append(probability)
    plot_images(similar_image_paths, distances, message)
```

This function is defined the book's Github website (see *http://PracticalDeepLearning.ai*), at *code/chapter-3*).

Now the fun starts! Which images are we most confident contain dogs? Let's find images with the highest prediction probability (i.e., closest to 1.0; see Figure 3-9) with the predicted class dog (i.e., 1):

```
# Most confident predictions of 'dog'
indices = get_images_with_sorted_probabilities(prediction_table,
get_highest_probability=True, label=1, number_of_items=10,
only_false_predictions=False)
message = 'Images with the highest probability of containing dogs'
display(indices[:10], message)
```

Figure 3-9. Images with the highest probability of containing dogs

These images are indeed very dog-like. One of the reasons the probability is so high may be the fact that the images contain multiple dogs, as well as clear, unambiguous views. Now let's try to find which images we are least confident contain dogs (see Figure 3-10):

```
# Least confident predictions of 'dog'
indices = get_images_with_sorted_probabilities(prediction_table,
get_highest_probability=False, label=1, number_of_items=10,
only_false_predictions=False)
message = 'Images with the lowest probability of containing dogs'
display(indices[:10], message)
```

Figure 3-10. Images with the lowest probability of containing dogs

To repeat, these are the images our classifier is most unsure of containing a dog. Most of these predictions are at the tipping point (i.e., 0.5 probability) to be the majority prediction. Keep in mind the probability of being a cat is just slightly smaller, around 0.49. Compared to the previous set of images, the animals appearing in these images are often smaller and less clear. And these images often result in mispredictions—only 2 of the 10 images were correctly predicted. One possible way to do better here is to train with a larger set of images.

If you are concerned about these misclassifications, worry not. A simple trick to improve the classification accuracy is to have a higher threshold for accepting a classifier's results, say 0.75. If the classifier is unsure of an image category, its results are withheld. In Chapter 5, we look at how to find an optimal threshold.

Speaking of mispredictions, they are obviously expected when the classifier has low confidence (i.e., near 0.5 probability for a two-class problem). But what we don't want is to mispredict when our classifier is really sure of its predictions. Let's check which images the classifier is confident contain dogs in spite of them being cats (see Figure 3-11):

```
# Incorrect predictions of 'dog'
indices = get_images_with_sorted_probabilities(prediction_table,
get_highest_probability=True, label=1, number_of_items=10,
only_false_predictions=True)
message = 'Images of cats with the highest probability of containing dogs'
display(indices[:10], message)
```

Figure 3-11. Images of cats with the highest probability of containing dogs

Hmm…turns out half of these images contain both cats and dogs, and our classifier is correctly predicting the dog category because they are bigger in size in these images. Thus, it's not the classifier but the data that is incorrect here. This often happens in large datasets. The remaining half often contains unclear and relatively smaller objects (but ideally we want lower confidence for these difficult-to-identify images).

Repeating the same set of questions for the cat class, which images are more cat-like (see Figure 3-12)?

```
# Most confident predictions of 'cat'
indices = get_images_with_sorted_probabilities(prediction_table,
get_highest_probability=True, label=0, number_of_items=10,
only_false_predictions=False)
message = 'Images with the highest probability of containing cats'
display(indices[:10], message)
```

Figure 3-12. Images with the highest probability of containing cats

Interestingly, many of these have multiple cats. This affirms our previous hypothesis that multiple clear, unambiguous views of cats can give higher probabilities. On the other hand, which images are we most unsure about containing cats (see Figure 3-13)?

```
# Least confident predictions of 'cat'
indices = get_images_with_sorted_probabilities(prediction_table,
get_highest_probability=False, label=0, number_of_items=10,
only_false_predictions=False)
message = 'Images with the lowest probability of containing cats'
display(indices[:10], message)
```

Figure 3-13. Images with the lowest probability of containing cats

As seen previously, the key object size is small, and some of the images are quite unclear, meaning that there is too much contrast in some cases or the object is too bright, something not in line with most of the training images. For example, the camera flash in the eighth (dog.6680) and tenth (dog.1625) images in Figure 3-13 makes the dog difficult to recognize. The sixth image contains a dog in front of a sofa of the same color. Two images contain cages.

Lastly, which images is our classifier mistakenly sure of containing cats (see Figure 3-14)?

```
# Incorrect predictions of 'cat'
indices = get_images_with_sorted_probabilities(prediction_table,
get_highest_probability=True, label=0, number_of_items=10,
only_false_predictions=True)
message = 'Images of dogs with the highest probability of containing cats'
display(indices[:10], message)
```

Figure 3-14. Images of dogs with the highest probability of containing cats

These mispredictions are what we want to reduce. Some of them are clearly wrong, whereas others are understandably confusing images. The sixth image (dog.4334) in Figure 3-14 seems to be incorrectly labeled as a dog. The seventh and tenth images are difficult to distinguish against the background. The first and tenth lack enough texture within them to give the classifier enough identification power. And some of the dogs are too small, like the second and fourth.

Going over the various analyses, we can summarize that mispredictions can be caused by low illumination, unclear, difficult-to-distinguish backgrounds, lack of texture, and smaller occupied area with regard to the image.

Analyzing our predictions is a great way to understand what our model has learned and what it's bad at, and highlights opportunities to enhance its predictive power. Increasing the size of the training examples and more robust augmentation will help in improving the classification. It's also important to note that showing real-world images to our model (images that look similar to the scenario where our app will be used) will help improve its accuracy drastically. In Chapter 5, we make the classifier more robust.

Figure 3-14. Images of dogs with the highest probability of containing cats

These mispredictions are what we want to reduce. Some of them are clearly wrong, whereas others are understandably confusing images. The sixth image (dog.4334) in Figure 3-14 seems to be incorrectly labeled as a dog. The seventh and tenth images are difficult to distinguish against the background. The first and tenth lack enough texture within them to give the classifier enough identification power. And some of the dogs are too small, like the second and fourth.

Going over the various analyses, we can summarize that mispredictions can be caused by low illumination, unclear, difficult-to-distinguish backgrounds, lack of texture, and smaller occupied area with regard to the image.

Analyzing our predictions is a great way to understand what our model has learned and what it’s bad at, and highlights opportunities to enhance its predictive power. Increasing the size of the training examples and more robust augmentation will help in improving the classification. It’s also important to note that showing real-world images to our model (images that look similar to the scenario where our app will be used) will help improve its accuracy drastically. In Chapter 5, we make the classifier more robust.

Further Reading

To help understand neural networks and CNNs better, our website features a learning guide which includes recommended resources like video lectures, blogs and, more interestingly, interactive visual tools which allow you to play with different scenarios in the browser without the need to install any packages. If you're a first-time learner of deep learning, we highly recommend this guide in order to strengthen your foundational knowledge. It covers the theory that you will need to build the intuition to solve future problems. We use Google's TensorFlow Playground (Figure 3-15) for neural networks and Andrej Karpathy's ConvNetJS (Figure 3-16) for CNNs.

Figure 3-15. Building a neural network in TensorFlow Playground

Figure 3-16. Defining a CNN and visualizing the output of each layer during training in ConvNetJS

We additionally have a short guide in Appendix A, which summarizes convolutional neural networks, as a ready reference.

Figure 3-16. Defining a CNN and visualizing the output of each layer during training in ConvNetJS

We additionally have a short guide in Appendix A, which summarizes convolutional neural networks, as a ready reference.

Summary

In this chapter, we introduced the concept of transfer learning. We reused a pretrained model to build our own cats versus dogs classifier in under 30 lines of code and with barely 500 images, reaching state-of-the-art accuracy in a few minutes. By writing this code, we also debunk the myth that we need millions of images and powerful GPUs to train our classifier (though they help).

Hopefully, with these skills, you might be able to finally answer the age-old question of who let the dogs out.

In the next couple of chapters, we use this learning to understand CNNs in more depth and take the model accuracy to the next level.

Summary

In this chapter, we introduced the concept of transfer learning. We reused a pretrained model to build our own cats versus dogs classifier in under 30 lines of code and with barely 500 images, reaching state-of-the-art accuracy in a few minutes. By writing this code, we also debunk the myth that we need millions of images and powerful GPUs to train our classifier (though they help).

Hopefully, with these skills, you might be able to finally answer the age-old question of who let the dogs out.

In the next couple of chapters, we use this learning to understand CNNs in more depth and take the model accuracy to the next level.

Chapter 4. Building a Reverse Image Search Engine: Understanding Embeddings

Bob just bought a new home and is looking to fill it up with some fancy modern furniture. He's flipping endlessly through furniture catalogs and visiting furniture showrooms, but hasn't yet landed on something he likes. Then one day, he spots the sofa of his dreams—a unique L-shaped white contemporary sofa in an office reception. The good news is that he knows what he wants. The bad news is that he doesn't know where to buy it from. The brand and model number is not written on the sofa. Asking the office manager doesn't help either. So, he takes a few photos from different angles to ask around in local furniture shops, but tough luck: no one knows this particular brand. And searching on the internet with keywords like "white L-shaped," "modern sofa" gives him thousands of results, but not the one he's looking for.

Alice hears Bob's frustration and asks, "Why don't you try reverse image search?" Bob uploads his images on Google and Bing's Reverse Image Search and quickly spots a similar-looking image on an online shopping website. Taking this more perfect image from the website, he does a few more reverse image searches and finds other websites offering the same sofa at cheaper prices. After a few minutes of being online, Bob has officially ordered his dream sofa!

Reverse image search (or as it is more technically known, *instance retrieval*) enables developers and researchers to build scenarios beyond simple keyword search. From discovering visually similar objects on Pinterest to recommending similar songs on Spotify to camera-based product search on Amazon, a similar class of technology under the hood is used. Sites like TinEye alert photographers on copyright infringement when their photographs are posted without consent on the internet. Even face recognition in several security systems uses a similar concept to ascertain the identity of the person.

The best part is, with the right knowledge, you can build a working replica of

many of these products in a few hours. So let's dig right in!

Here's what we're doing in this chapter:

1. Performing feature extraction and similarity search on Caltech101 and Caltech256 datasets
2. Learning how to scale to large datasets (up to billions of images)
3. Making the system more accurate and optimized
4. Analyzing case studies to see how these concepts are used in mainstream products

many of these products in a few hours. So let’s dig right in!

Here’s what we’re doing in this chapter:

1. Performing feature extraction and similarity search on Caltech101 and Caltech256 datasets
2. Learning how to scale to large datasets (up to billions of images)
3. Making the system more accurate and optimized
4. Analyzing case studies to see how these concepts are used in mainstream products

Image Similarity

The first and foremost question is: given two images, are they similar or not?

There are several approaches to this problem. One approach is to compare patches of areas between two images. Although this can help find exact or near-exact images (that might have been cropped), even a slight rotation would result in dissimilarity. By storing the hashes of the patches, duplicates of an image can be found. One use case for this approach would be the identification of plagiarism in photographs.

Another naive approach is to calculate the histogram of RGB values and compare their similarities. This might help find near-similar images captured in the same environment without much change in the contents. For example, in Figure 4-1, this technique is used in image deduplication software aimed at finding bursts of photographs on your hard disk, so you can select the best one and delete the rest. Of course, there is an increasing possibility of false positives as your dataset grows. Another downside to this approach is that small changes to the color, hue, or white balance would make recognition more difficult.

Figure 4-1. RGB histogram-based "Similar Image Detector" program

A more robust traditional computer vision-based approach is to find visual features near edges using algorithms like Scale-Invariant Feature Transform (SIFT), Speeded Up Robust Features (SURF), and Oriented FAST and Rotated BRIEF (ORB) and then compare the number of similar features that are common between the two photos. This helps you go from a generic image-level understanding to a relatively robust object-level understanding. Although this is great for images with rigid objects that have less variation like the printed sides of a box of cereal, which almost always look the same, it's less helpful for comparing deformable objects like humans and animals, which can exhibit different poses. As an example, you can see the features being displayed on the camera-based product search experience on the Amazon app. The app displays these features in the form of blue dots (Figure 4-2). When it sees a sufficient number of features, it sends them to the Amazon servers to retrieve product information.

Figure 4-2. Product scanner in Amazon app with visual features highlighted

Going deeper, another approach is to find the category (e.g., sofa) of an image using deep learning and then find other images within the same category. This is equivalent to extracting metadata from an image so that it can then be indexed and used in a typical text query-based search. This can be easily scaled by using the metadata in open source search engines like ElasticSearch. Many ecommerce sites show recommendations based on tags extracted from an image while performing a query-based search internally. As you would expect, by extracting the tags, we lose certain information like color, pose, relationships between

objects in the scene, and so on. Additionally, a major disadvantage of this approach is that it requires enormous volumes of labeled data to train the classifier for extracting these labels on new images. And every time a new category needs to be added, the model needs to be retrained.

Because our aim is to search among millions of images, what we ideally need is a way to summarize the information contained in the millions of pixels in an image into a smaller representation (of say a few thousand dimensions), and have this summarized representation be close together for similar objects and further away for dissimilar items.

Luckily, deep neural networks come to the rescue. As we saw in Chapter 2 and Chapter 3, the CNNs take an image input and convert it into feature vectors of a thousand dimensions, which then act as input to a classifier that outputs the top identities to which the image might belong (say dog or cat). The *feature vectors* (also called *embeddings* or *bottleneck features*) are essentially a collection of a few thousand floating-point values. Going through the convolution and pooling layers in a CNN is basically an act of reduction, to filter the information contained in the image to its most important and salient constituents, which in turn form the bottleneck features. Training the CNN molds these values in such a way that items belonging to the same class have small Euclidean distance between them (or simply the square root of the sum of squares of the difference between corresponding values) and items from different classes are separated by larger distances. This is an important property that helps solve so many problems where a classifier can't be used, especially in unsupervised problems because of a lack of adequate labeled data.

TIP

An ideal way to find similar images would be to use *transfer learning*. For example, pass the images through a pretrained convolutional neural network like ResNet-50, extract the features, and then use a metric to calculate the error rate like the Euclidean distance.

Enough talk, let's code!

a

s

Feature Extraction

An image is worth a thousand ~~words~~ features.

In this section, we play with and understand the concepts of feature extraction, primarily with the Caltech 101 dataset (131 MB, approximately 9,000 images), and then eventually with Caltech 256 (1.2 GB, approximately 30,000 images). Caltech 101, as the name suggests, consists of roughly 9,000 images in 101 categories, with about 40 to 800 images per category. It's important to note that there is a 102nd category called "BACKGROUND_Google" consisting of random images not contained in the first 101 categories, which needs to be deleted before we begin experimenting. Remember that all of the code we are writing is also available in the GitHub repository.

Let's download the dataset:

```
$ wget
http://www.vision.caltech.edu/Image_Datasets/Caltech101/
101_ObjectCategories.tar.gz
$ tar -xvf 101_ObjectCategories.tar.gz
$ mv 101_ObjectCategories caltech101
$ rm -rf caltech101/BACKGROUND_Google
```

Now, import all of the necessary modules:

```
import numpy as np
from numpy.linalg import norm
import pickle
from tqdm import tqdm, tqdm_notebook
import os
import time
from tf.keras.preprocessing import image
from tf.keras.applications.resnet50 import ResNet50, preprocess_input
```

Load the ResNet-50 model without the top classification layers, so we get only the *bottleneck features.* Then define a function that takes an image path, loads the image, resizes it to proper dimensions supported by ResNet-50, extracts the features, and then normalizes them:

```
model = ResNet50(weights='imagenet', include_top=False,
```

```
                 input_shape=(224, 224, 3))
def extract_features(img_path, model):
    input_shape = (224, 224, 3)
    img = image.load_img(img_path, target_size=(
        input_shape[0], input_shape[1]))
    img_array = image.img_to_array(img)
    expanded_img_array = np.expand_dims(img_array, axis=0)
    preprocessed_img = preprocess_input(expanded_img_array)
    features = model.predict(preprocessed_img)
    flattened_features = features.flatten()
    normalized_features = flattened_features / norm(flattened_features)
    return normalized_features
```

TIP

The function defined in the previous example is the `key` function that we use for almost every feature extraction need in Keras.

That's it! Let's see the feature-length that the model generates:

```
features = extract_features('../../sample_images/cat.jpg', model)
print(len(features))
```

annoy

```
> 2048
```

The ResNet-50 model generated 2,048 features from the provided image. Each feature is a floating-point value between 0 and 1.

TIP

If your model is trained or fine tuned on a dataset that is not similar to ImageNet, redefine the "preprocess_input(img)" step accordingly. The mean values used in the function are particular to the ImageNet dataset. Each model in Keras has its own preprocessing function so make sure you are using the right one.

Now it's time to extract features for the entire dataset. First, we get all the filenames with this handy function, which recursively looks for all the image

files (defined by their extensions) under a directory:

```
extensions = ['.jpg', '.JPG', '.jpeg', '.JPEG', '.png', '.PNG']
def get_file_list(root_dir):
    file_list = []
    counter = 1
    for root, directories, filenames in os.walk(root_dir):
        for filename in filenames:
            if any(ext in filename for ext in extensions):
                file_list.append(os.path.join(root, filename))
                counter += 1
    return file_list
```

Then, we provide the path to our dataset and call the function:

```
# path to the datasets
root_dir = '../../datasets/caltech101'
filenames = sorted(get_file_list(root_dir))
```

We now define a variable that will store all of the features, go through all filenames in the dataset, extract their features, and append them to the previously defined variable:

```
feature_list = []
for i in tqdm_notebook(range(len(filenames))):
    feature_list.append(extract_features(filenames[i], model))
```

On a CPU, this should take under an hour. On a GPU, only a few minutes.

TIP

To get a better sense of time, use the super handy tool `tqdm`, which shows a progress meter (Figure 4-3) along with the speed per iteration as well as the time that has passed and expected finishing time. In Python, wrap an iterable with `tqdm`; for example, `tqdm(range(10))`. Its Jupyter Notebook variant is `tqdm_notebook`.

Figure 4-3. Progress bar shown with `tqdm_notebook`

Finally, write these features to a pickle file so that we can use them in the future

without having to recalculate them:

```
pickle.dump(feature_list, open('data/features-caltech101-resnet.pickle', 'wb'))
pickle.dump(filenames, open('data/filenames-caltech101.pickle','wb'))
```

That’s all folks! We’re done with the feature extraction part.

without having to recalculate them:

```
pickle.dump(feature_list, open('data/features-caltech101-resnet.pickle', 'wb'))
pickle.dump(filenames, open('data/filenames-caltech101.pickle','wb'))
```

That’s all folks! We’re done with the feature extraction part.

Similarity Search

Given a photograph, our aim is to find another photo in our dataset similar to the current one. We begin by loading the precomputed features:

```
filenames = pickle.load(open('data/filenames-caltech101.pickle', 'rb'))
feature_list = pickle.load(open('data/features-caltech101-resnet.pickle', 'rb'))
```

We'll use Python's machine learning library `scikit-learn` for finding *nearest neighbors* of the query features; that is, features that represent a query image. We train a nearest-neighbor model using the brute-force algorithm to find the nearest five neighbors based on Euclidean distance (to install `scikit-learn` on your system, use `pip3 install sklearn`):

```
from sklearn.neighbors import NearestNeighbors
neighbors = NearestNeighbors(n_neighbors=5, algorithm='brute',
metric='euclidean').fit(feature_list)
distances, indices = neighbors.kneighbors([feature_list[0]])
```

Now you have both the indices and distances of the nearest five neighbors of the very first query feature (which represents the first image). Notice the quick execution of the first step—the training step. Unlike training most machine learning models, which can take from several minutes to hours on large datasets, instantiating the nearest-neighbor model is instantaneous because at training time there isn't much processing. This is also called *lazy learning* because all the processing is deferred to classification or inference time.

Now that we know the indices, let's see the actual image behind that feature. First, we pick an image to query, located at say, index = 0:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline       # Show the plots as a cell within the Jupyter Notebooks
plt.imshow(mpimg.imread(filenames[0]))
```

Figure 4-4 shows the result.

Figure 4-4. The query image from the Caltech-101 dataset

Now, let's examine the nearest neighbors by plotting the first result.

```
plt.imshow(mpimg.imread(filenames[indices[0]]))
```

Figure 4-5 shows that result.

Figure 4-5. The nearest neighbor to our query image

Wait, isn't that a duplicate? Actually, the nearest index will be the image itself because that is what is being queried:

```
for i in range(5):
    print(distances[0][i])
```

```
0.0
0.8285478
0.849847
0.8529018
```

This is also confirmed by the fact that the distance of the first result is zero. Now

let's plot the real first nearest neighbor:

```
plt.imshow(mpimg.imread(filenames[indices[1]]))
```

Take a look at the result this time in Figure 4-6.

Figure 4-6. The second nearest neighbor of the queried image

This definitely looks like a similar image. It captured a similar concept, has the same image category (faces), same gender, and similar background with pillars and vegetation. In fact, it's the same person!

We would probably use this functionality regularly, so we have already built a helper function `plot_images()` that visualizes several query images with their nearest neighbors. Now let's call this function to visualize the nearest neighbors of six random images. Also, note that every time you run the following piece of code, the displayed images will be different (Figure 4-7) because the displayed images are indexed by a random integer.

```
for i in range(6):
    random_image_index = random.randint(0,num_images)
    distances, indices = neighbors.kneighbors([featureList[random_image_index]])
    # don't take the first closest image as it will be the same image
    similar_image_paths = [filenames[random_image_index]] +
                          [filenames[indices[0][i]] for i in range(1,4)]
    plot_images(similar_image_paths, distances[0])
```

Figure 4-7. Nearest neighbor for different images returns similar-looking images

Visualizing Image Clusters with t-SNE

Let's step up the game by visualizing the entire dataset!

To do this, we need to reduce the dimensions of the feature vectors because it's not possible to plot a 2,048-dimension vector (the feature-length) in two dimensions (the paper). The t-distributed stochastic neighbor embedding (t-SNE algorithm reduces the high-dimensional feature vector to 2D, providing a bird's-eye view of the dataset, which is helpful in recognizing clusters and nearby images. t-SNE is difficult to scale to large datasets, so it is a good idea to reduce the dimensionality using Principal Component Analysis (PCA) and then call t-SNE:

```
# Perform PCA over the features
num_feature_dimensions=100      # Set the number of features
pca = PCA(n_components = num_feature_dimensions)
pca.fit(featureList)
feature_list_compressed = pca.transform(featureList)

# For speed and clarity, we'll analyze about first half of the dataset.
selected_features = feature_list_compressed[:4000]
selected_class_ids = class_ids[:4000]
selected_filenames = filenames[:4000]

tsne_results =
TSNE(n_components=2,verbose=1,metric='euclidean')
    .fit_transform(selected_features)

# Plot a scatter plot from the generated t-SNE results
colormap = plt.cm.get_cmap('coolwarm')
scatter_plot = plt.scatter(tsne_results[:,0],tsne_results[:,1], c =
               selected_class_ids, cmap=colormap)
plt.colorbar(scatter_plot)
plt.show()
```

We discuss PCA in more detail in later sections. In order to scale to larger dimensions, use Uniform Manifold Approximation and Projection (UMAP).

Figure 4-8 shows clusters of similar classes, and how they are spread close to one another.

)

Figure 4-8. t-SNE visualizing clusters of image features, where each cluster represents one object class in the same color

Each color in Figure 4-8 indicates a different class. To make it even more clear, we can use another helper function, `plot_images_in_2d()`, to plot the images in these clusters, as demonstrated in Figure 4-9.

Figure 4-9. t-SNE visualization showing image clusters; similar images are in the same cluster

Neat! There is a clearly demarcated cluster of human faces, flowers, vintage cars, ships, bikes, and a somewhat spread-out cluster of land and marine animals There are lots of images on top of one another, which makes Figure 4-9 a tad bit confusing, so let's try to plot the t-SNE as clear tiles with the helper function `tsne_to_grid_plotter_manual()`, the results of which you can see in Figure 4-10.

```
tsne_to_grid_plotter_manual(tsne_results[:,0], tsne_results[:,1],
                            selected_filenames)
```

Figure 4-10. t-SNE visualization with tiled images; similar images are close together

This is definitely much clearer. We can see similar images are colocated within the clusters of human faces, chairs, bikes, airplanes, ships, laptops, animals, watches, flowers, tilted minarets, vintage cars, anchor signs, and cameras, all close to their own kind. Birds of a feather indeed do flock together!

TIP

2D clusters are great, but visualizing them in 3D would look stellar. It would be even better if they could be rotated, zoomed in and out, and manipulated using the mouse without any coding. And bonus points if the data could be searched interactively, revealing its neighbors. The TensorFlow Embedding projector does all this and more in a browser-based GUI tool. The preloaded embeddings from image and text

datasets are helpful in getting a better intuition of the power of embeddings. And, as Figure 4-11 shows, it's reassuring to see deep learning figure out that John Lennon, Led Zeppelin, and Eric Clapton happen to be used in a similar context to the Beatles in the English language.

Figure 4-11. TensorFlow Embedding projector showing a 3D representation of 10,000 common English words and highlighting words related to "Beatles"

datasets are helpful in getting a better intuition of the power of embeddings. And, as Figure 4-11 shows, it's reassuring to see deep learning figure out that John Lennon, Led Zeppelin, and Eric Clapton happen to be used in a similar context to the Beatles in the English language.

Figure 4-11. TensorFlow Embedding projector showing a 3D representation of 10,000 common English words and highlighting words related to "Beatles"

Improving the Speed of Similarity Search

There are several opportunities to improve the speed of the similarity search step. For similarity search, we can make use of two strategies: either reduce the feature-length, or use a better algorithm to search among the features. Let’s examine each of these strategies individually.

Improving the Speed of Similarity Search

There are several opportunities to improve the speed of the similarity search step. For similarity search, we can make use of two strategies: either reduce the feature-length, or use a better algorithm to search among the features. Let’s examine each of these strategies individually.

Length of Feature Vectors

Ideally, we would expect that the smaller the amount of data in which to search, the faster the search should be. Recall that the ResNet-50 model gives 2,048 features. With each feature being a 32-bit floating-point, each image is represented by an 8 KB feature vector. For a million images, that equates to nearly 8 GB. Imagine how slow it would be to search among 8 GB worth of features. To give us a better picture of our scenario, Table 4-1 gives the feature-lengths that we get from different models.

Table 4-1. Top 1% accuracy and feature-lengths for different CNN models

Model	Bottleneck feature-length	Top-1% accuracy on ImageNet
VGG16	512	71.5%
VGG19	512	72.7%
MobileNet	1024	66.5%
InceptionV3	2048	78.8%
ResNet-50	2048	75.9%
Xception	2048	79.0%

NOTE

Under the hood, many models available in `tf.keras.applications` yield several thousand features. For example, InceptionV3 yields features in the shape of 1 x 5 x 5 x 2048, which translates to 2,048 feature maps of 5 x 5 convolutions, resulting in a total of 51,200 features. Hence, it becomes essential to reduce this large vector by using an average or max-pooling layer. The pooling layer will condense each convolution (e.g., 5 x 5 layer) into a single value. This can be defined during model instantiation as follows:

```
model = InceptionV3(weights='imagenet', include_top=False,
input_shape = (224,224,3), pooling='max')
```

For models that yield a large number of features, you will usually find that all code examples make use of this pooling option. Table 4-2 shows the before and after effect of max pooling on the number of features in different models.

Table 4-2. Number of features before and after pooling for different models

Model	# features before pooling	# features after pooling
ResNet-50	[1,1,1,2048] = 2048	2048
InceptionV3	[1,5,5,2048] = 51200	2048
MobileNet	[1,7,7,1024] = 50176	1024

As we can see, almost all the models generate a large number of features. Imagine how much faster the search would be if we could reduce to a mere 100 features (a whopping reduction of 10 to 20 times!) without compromising the quality of the results. Apart from just the size, this is an even bigger improvement for big data scenarios, for which the data can be loaded into RAM all at once instead of periodically loading parts of it, thus giving an even bigger speedup. PCA will help us make this happen.

Table 4-2. Number of features before and after pooling for different models

Model	# features before pooling	# features after pooling
ResNet-50	[1,1,1,2048] = 2048	2048
InceptionV3	[1,5,5,2048] = 51200	2048
MobileNet	[1,7,7,1024] = 50176	1024

As we can see, almost all the models generate a large number of features. Imagine how much faster the search would be if we could reduce to a mere 100 features (a whopping reduction of 10 to 20 times!) without compromising the quality of the results. Apart from just the size, this is an even bigger improvement for big data scenarios, for which the data can be loaded into RAM all at once instead of periodically loading parts of it, thus giving an even bigger speedup. PCA will help us make this happen.

Reducing Feature-Length with PCA

PCA is a statistical procedure that questions whether features representing the data are equally important. Are some of the features redundant enough that we can get similar classification results even after removing those features? PCA is considered one of the go-to techniques for dimensionality reduction. Note that it does not eliminate redundant features; rather, it generates a new set of features that are a linear combination of the input features. These linear features are orthogonal to one another, which is why all the redundant features are absent. These features are known as *principal components.*

Performing PCA is pretty simple. Using the `scikit-learn` library, execute the following:

```
import sklearn.decomposition.PCA as PCA
num_feature_dimensions=100
pca = PCA(n_components = num_feature_dimensions)
pca.fit(feature_list)
feature_list_compressed = pca.transform(feature_list)
```

PCA can also tell us the relative importance of each feature. The very first dimension has the most variance and the variance keeps on decreasing as we go on:

```
# Explain the importance of first 20 features
print(pca.explained_variance_ratio_[0:20])
```

```
[ 0.07320023  0.05273142   0.04310822 0.03494248  0.02166119  0.0205037
  0.01974325  0.01739547   0.01611573 0.01548918  0.01450421  0.01311005
  0.01200541  0.0113084    0.01103872 0.00990405  0.00973481  0.00929487
  0.00915592  0.0089256 ]
```

Hmm, why did we pick 100 dimensions from the original 2,048? Why not 200? PCA is representing our original feature vector but in reduced dimensions. Each new dimension has diminishing returns in representing the original vector (i.e., the new dimension might not explain the data much) and takes up valuable space. We can balance between how well the original data is explained versus how much we want to reduce it. Let’s visualize the importance of say the first

200 dimensions.

```
pca = PCA(200)
pca.fit(feature_list)
matplotlib.style.use('seaborn')
plt.plot(range(1,201),pca.explained_variance_ratio_,'o--', markersize=4)
plt.title ('Variance for each PCA dimension')
plt.xlabel('PCA Dimensions')
plt.ylabel('Variance')
plt.grid(True)
plt.show()
```

Figure 4-12 presents the results.

Figure 4-12. Variance for each PCA dimension

The individual variance will tell us how important the newly added features are. For example, after the first 100 dimensions, the additional dimensions don't add much variance (almost equal to 0) and can be neglected. Without even checking the accuracy it is safe to assume that the PCA with 100 dimensions will be a robust model. Another way to look at this is to visualize how much of the

original data is explained by the limited number of features by finding the cumulative variance (see Figure 4-13).

```
plt.plot(range(1,201),pca.explained_variance_ratio_.cumsum(),'o--', markersize=4)
plt.title ('Cumulative Variance with each PCA dimension')
plt.xlabel('PCA Dimensions')
plt.ylabel('Variance')
plt.grid(True)
plt.show()
```

Figure 4-13. Cumulative variance with each PCA dimension

As expected, adding 100 dimensions (from 100 to 200) adds only 0.1 variance and begins to gradually plateau. For reference, using the full 2,048 features would result in a cumulative variance of 1.

The number of dimensions in PCA is an important parameter that we can tune to the problem at hand. One way to directly justify a good threshold is to find a good balance between the number of features and its effect on accuracy versus speed:

```
pca_dimensions = [1,2,3,4,5,10,20,50,75,100,150,200]
pca_accuracy = []
pca_time = []

for dimensions in pca_dimensions:
    # Perform PCA
    pca = PCA(n_components = dimensions)
    pca.fit(feature_list)
    feature_list_compressed = pca.transform(feature_list[:])
    # Calculate accuracy over the compressed features
    accuracy, time_taken = accuracy_calculator(feature_list_compressed[:])
    pca_time.append(time_taken)
    pca_accuracy.append(accuracy)
    print("For PCA Dimensions = ", dimensions, ",\tAccuracy = ",accuracy,"%",
",\tTime = ", pca_time[-1])
```

We visualize these results using the graph in Figure 4-14 and see that after a certain number of dimensions an increase in dimensions does not lead to higher accuracy:

```
plt.plot(pca_time, pca_accuracy,'o--', markersize=4)
for label, x, y in zip(pca_dimensions, pca_time,pca_accuracy):
    plt.annotate(label, xy=(x, y), ha='right', va='bottom')
plt.title ('Test Time vs Accuracy for each PCA dimension')
plt.xlabel('Test Time')
plt.ylabel('Accuracy')
plt.grid(True)
plt.show()
```

Figure 4-14. Test time versus accuracy for each PCA dimension

As is visible in the graph, there is little improvement in accuracy after increasing beyond a feature-length of 100 dimensions. With almost 20 times fewer dimensions (100) than the original (2,048), this offers drastically higher speed and less time on almost any search algorithm, while achieving similar (and sometimes slightly better) accuracy. Hence, 100 would be an ideal feature-length for this dataset. This also means that the first 100 dimensions contain the most information about the dataset.

There are a number of benefits to using this reduced representation, like efficient use of computational resources, noise removal, better generalization due to fewer dimensions, and improved performance for machine learning algorithms learning on this data. By reducing the distance calculation to the most important features, we can also improve the result accuracy slightly. This is because previously all the 2,048 features were contributing equally in the distance calculation, whereas now, only the most important 100 features get their say. But, more importantly, it saves us from the *curse of dimensionality*. It's observed that as the number of dimensions increases, the ratio of the Euclidean distance between the two closest points and the two furthest points tends to become 1. In very high-dimensional space, the majority of points from a real-world dataset seem to be a similar distance away from one another, and the Euclidean distance

metric begins to fail in discerning similar versus dissimilar items. PCA helps bring sanity back.

You can also experiment with different distances like Minkowski distance, Manhattan distance, Jaccardian distance, and weighted Euclidean distance (where the weight is the contribution of each feature as explained in `pca.explained_variance_ratio_`).

Now, let’s turn our minds toward using this reduced set of features to make our search even faster.

metric begins to fail in discerning similar versus dissimilar items. PCA helps bring sanity back.

You can also experiment with different distances like Minkowski distance, Manhattan distance, Jaccardian distance, and weighted Euclidean distance (where the weight is the contribution of each feature as explained in `pca.explained_variance_ratio_`).

Now, let’s turn our minds toward using this reduced set of features to make our search even faster.

Scaling Similarity Search with Approximate Nearest Neighbors

What do we want? Nearest neighbors. What is our baseline? Brute-force search. Although convenient to implement in two lines, it goes over each element and hence scales linearly with data size (number of items as well as the number of dimensions). Having PCA take our feature vector from a length of 2,048 to 100 will not only yield a 20-times reduction in data size, but also result in an increase in speed of 20 times when using brute force. PCA does pay off!

Let's assume similarity searching a small collection of 10,000 images, now represented with 100 feature-length vectors, takes approximately 1 ms. Even though this looks fast for 10,000 items, in a real production system with larger data, perhaps 10 million items, this will take more than a second to search. Our system might not be able to fulfill more than one query per second per CPU core. If you receive 100 requests per second from users, even running on multiple CPU cores of the machine (and loading the search index per thread), you would need multiple machines to be able to serve the traffic. In other words, an inefficient algorithm means money, lots of money, spent on hardware.

Brute force is our baseline for every comparison. As in most algorithmic approaches, brute force is the slowest approach. Now that we have our baseline set, we will explore approximate nearest-neighbor algorithms. Instead of guaranteeing the correct result as with the brute-force approach, approximation algorithms *generally* get the correct result because they are...well, approximations. Most of the algorithms offer some form of tuning to balance between correctness and speed. It is possible to evaluate the quality of the results by comparing against the results of the brute-force baseline.

Scaling Similarity Search with Approximate Nearest Neighbors

What do we want? Nearest neighbors. What is our baseline? Brute-force search. Although convenient to implement in two lines, it goes over each element and hence scales linearly with data size (number of items as well as the number of dimensions). Having PCA take our feature vector from a length of 2,048 to 100 will not only yield a 20-times reduction in data size, but also result in an increase in speed of 20 times when using brute force. PCA does pay off!

Let's assume similarity searching a small collection of 10,000 images, now represented with 100 feature-length vectors, takes approximately 1 ms. Even though this looks fast for 10,000 items, in a real production system with larger data, perhaps 10 million items, this will take more than a second to search. Our system might not be able to fulfill more than one query per second per CPU core. If you receive 100 requests per second from users, even running on multiple CPU cores of the machine (and loading the search index per thread), you would need multiple machines to be able to serve the traffic. In other words, an inefficient algorithm means money, lots of money, spent on hardware.

Brute force is our baseline for every comparison. As in most algorithmic approaches, brute force is the slowest approach. Now that we have our baseline set, we will explore approximate nearest-neighbor algorithms. Instead of guaranteeing the correct result as with the brute-force approach, approximation algorithms *generally* get the correct result because they are...well, approximations. Most of the algorithms offer some form of tuning to balance between correctness and speed. It is possible to evaluate the quality of the results by comparing against the results of the brute-force baseline.

Approximate Nearest-Neighbor Benchmark

There are several approximate nearest-neighbor (ANN) libraries out there, including well-known ones like Spotify's Annoy, FLANN, Facebook's Faiss, Yahoo's NGT, and NMSLIB. Benchmarking each of them would be a tedious task (assuming you get past installing some of them). Luckily, the good folks at *ann-benchmarks.com* (Martin Aumueller, Erik Bernhardsson, and Alec Faitfull) have done the legwork for us in the form of reproducible benchmarks on 19 libraries on large public datasets. We'll pick the comparisons on a dataset of feature embeddings representing words (instead of images) called GloVe. This 350 MB dataset consists of 400,000 feature vectors representing words in 100 dimensions. Figure 4-15 showcases their raw performance when tuned for correctness. Performance is measured in the library's ability to respond to queries each second. Recall that a measure of correctness is the fraction of top-*n* closest items returned with respect to the real top-*n* closest items. This ground truth is measured by brute-force search.

Figure 4-15. Comparison of ANN libraries (data from ann-benchmarks.com)

The strongest performers on this dataset return close to several thousand queries per second at the acceptable 0.8 recall. To put this in perspective, our brute-force search performs under 1 query per second. At the fastest, some of these libraries (like NGT) can return north of 15,000 results per second (albeit at a low recall, making it impractical for usage).

The strongest performers on this dataset return close to several thousand queries per second at the acceptable 0.8 recall. To put this in perspective, our brute-force search performs under 1 query per second. At the fastest, some of these libraries (like NGT) can return north of 15,000 results per second (albeit at a low recall, making it impractical for usage).

Which Library Should I Use?

It goes without saying that the library you use will end up depending heavily on your scenario. Each library presents a trade-off between search speed, accuracy, size of index, memory consumption, hardware use (CPU/GPU), and ease of setup. Table 4-3 presents a synopsis of different scenarios and recommendations as to which library might be work best for each scenario.

Table 4-3. ANN library recommendations

Scenario	Recommendation
I want to experiment quickly in Python without too much setup but I also care about fast speed.	Use Annoy or NMSLIB
I have a large dataset (up to 10 million entries or several thousand dimensions) and care utmost about speed.	Use NGT
I have a ridiculously large dataset (100 million-plus entries) and have a cluster of GPUs, too.	Use Faiss
I want to set a ground-truth baseline with 100% correctness. Then immediately move to a faster library, impress my boss with the orders of magnitude speedup, and get a bonus.	Use brute-force approach

We offer much more detailed examples in code of several libraries on the book's GitHub website (see *http://PracticalDeepLearning.ai*), but for our purposes here, we'll showcase our go-to library, Annoy, in detail and compare it with brute-force search on a synthetic dataset. Additionally, we briefly touch on Faiss and NGT.

Which Library Should I Use?

It goes without saying that the library you use will end up depending heavily on your scenario. Each library presents a trade-off between search speed, accuracy, size of index, memory consumption, hardware use (CPU/GPU), and ease of setup. Table 4-3 presents a synopsis of different scenarios and recommendations as to which library might be work best for each scenario.

Table 4-3. ANN library recommendations

Scenario	Recommendation
I want to experiment quickly in Python without too much setup but I also care about fast speed.	Use Annoy or NMSLIB
I have a large dataset (up to 10 million entries or several thousand dimensions) and care utmost about speed.	Use NGT
I have a ridiculously large dataset (100 million-plus entries) and have a cluster of GPUs, too.	Use Faiss
I want to set a ground-truth baseline with 100% correctness. Then immediately move to a faster library, impress my boss with the orders of magnitude speedup, and get a bonus.	Use brute-force approach

We offer much more detailed examples in code of several libraries on the book's GitHub website (see *http://PracticalDeepLearning.ai*), but for our purposes here, we'll showcase our go-to library, Annoy, in detail and compare it with brute-force search on a synthetic dataset. Additionally, we briefly touch on Faiss and NGT.

Creating a Synthetic Dataset

To make an apples-to-apples comparison between different libraries, we first create a million-item dataset composed of random floating-point values with mean 0 and variance 1. Additionally, we pick a random feature vector as our query to find the nearest neighbors:

```
num_items = 1000000
num_dimensions = 100
dataset = np.random.randn(num_items, num_dimensions)
dataset /= np.linalg.norm(dataset, axis=1).reshape(-1, 1)

random_index = random.randint(0,num_items)
query = dataset[random_index]
```

Creating a Synthetic Dataset

To make an apples-to-apples comparison between different libraries, we first create a million-item dataset composed of random floating-point values with mean 0 and variance 1. Additionally, we pick a random feature vector as our query to find the nearest neighbors:

```
num_items = 1000000
num_dimensions = 100
dataset = np.random.randn(num_items, num_dimensions)
dataset /= np.linalg.norm(dataset, axis=1).reshape(-1, 1)

random_index = random.randint(0,num_items)
query = dataset[random_index]
```

Brute Force

First, we calculate the time for searching with the brute-force algorithm. It goes through the entire data serially, calculating the distance between the query and current item one at a time. We use the `timeit` command for calculating the time First, we create the search index to retrieve the five nearest neighbors and then search with a query:

```
neighbors = NearestNeighbors(n_neighbors=5, algorithm='brute',
metric='euclidean').fit(dataset)
%timeit distances, indices = neighbors.kneighbors([query])

> 177 ms ± 136 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

TIP

The `timeit` command is a handy tool. To benchmark the time of a single operation, prefix it with this command. Compared to the time command, which runs a statement for one time, `timeit` runs the subsequent line multiple times to give more precise aggregated statistics along with the standard deviation. By default, it turns off garbage collection, making independent timings more comparable. That said, this might not reflect timings in real production loads where garbage collection is turned on.

Brute Force

First, we calculate the time for searching with the brute-force algorithm. It goes through the entire data serially, calculating the distance between the query and current item one at a time. We use the `timeit` command for calculating the time. First, we create the search index to retrieve the five nearest neighbors and then search with a query:

```
neighbors = NearestNeighbors(n_neighbors=5, algorithm='brute',
metric='euclidean').fit(dataset)
%timeit distances, indices = neighbors.kneighbors([query])

> 177 ms ± 136 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

TIP

The `timeit` command is a handy tool. To benchmark the time of a single operation, prefix it with this command. Compared to the time command, which runs a statement for one time, `timeit` runs the subsequent line multiple times to give more precise aggregated statistics along with the standard deviation. By default, it turns off garbage collection, making independent timings more comparable. That said, this might not reflect timings in real production loads where garbage collection is turned on.

Annoy

Annoy (Approximate Nearest Neighbors Oh Yeah) is a C++ library with Python bindings for searching nearest neighbors. Synonymous with speed, it was released by Spotify and is used in production to serve its music recommendations. In contrast to its name, it's actually fun and easy to use.

To use Annoy, we install it using `pip`:

```
$ pip install annoy
```

It's fairly straightforward to use. First, we build a search index with two hyperparameters: the number of dimensions of the dataset and the number of trees:

```
from annoy import AnnoyIndex
annoy_index = AnnoyIndex(num_dimensions) # Length of item vector that will be
indexed
for i in range(num_items):
    annoy_index.add_item(i, dataset[i])
annoy_index.build(40) # 40 trees
```

Now let's find out the time it takes to search the five nearest neighbors of one image:

```
%timeit indexes=t.get_nns_by_vector(query, 5, include_distances=True)

> 34.9 µs ± 165 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Now that is blazing fast! To put this in perspective, even for our million-item dataset, this can serve almost 28,000 requests on a single CPU core. Considering most CPUs have multiple cores, it should be able to handle more than 100,000 requests on a single system. The best part is that it lets you share the same index in memory between multiple processes. Thus, the biggest index can be equivalent to the size of your overall RAM, making it possible to serve multiple requests on a single system.

Other benefits include that it generates a modestly sized index. Moreover, it

decouples creating indexes from loading them, so you can create an index on one machine, pass it around, and then on your serving machine load it in memory and serve it.

TIP

Wondering about how many trees to use? More trees give higher precision, but larger indexes. Usually, no more than 50 trees are required to attain the highest precision.

decouples creating indexes from loading them, so you can create an index on one machine, pass it around, and then on your serving machine load it in memory and serve it.

TIP

Wondering about how many trees to use? More trees give higher precision, but larger indexes. Usually, no more than 50 trees are required to attain the highest precision.

NGT

Yahoo Japan's Neighborhood Graph and Tree (NGT) library currently leads most benchmarks and is best suited for large datasets (in millions of items) with large dimensions (in several thousands). Although the library has existed since 2016, its real entry into the industry benchmark scene happened in 2018 with the implementation of the ONNG algorithm (short for Optimization of indexing based on *k*-Nearest Neighbor Graph for proximity). Considering multiple threads might be running NGT on a server, it can place the index in shared memory with the help of memory mapped files, helping to reduce memory usage as well as increase load time.

NGT

Yahoo Japan's Neighborhood Graph and Tree (NGT) library currently leads most benchmarks and is best suited for large datasets (in millions of items) with large dimensions (in several thousands). Although the library has existed since 2016, its real entry into the industry benchmark scene happened in 2018 with the implementation of the ONNG algorithm (short for Optimization of indexing based on *k*-Nearest Neighbor Graph for proximity). Considering multiple threads might be running NGT on a server, it can place the index in shared memory with the help of memory mapped files, helping to reduce memory usage as well as increase load time.

Faiss

Faiss is Facebook's efficient similarity search library. It can scale to billions of vectors in RAM on a single server by storing a compressed representation of the vectors (compact quantization codes) instead of the original values. It's especially suited for dense vectors. It shines particularly on machines with GPUs by storing the index on GPU memory (VRAM). This works on both single-GPU and multi-GPU setups. It provides the ability to configure performance based on search time, accuracy, memory usage, and indexing time. It's one of the fastest known implementations of ANN search on GPU. Hey, if it's good enough for Facebook, it's good enough for most of us (as long as we have enough data).

While showing the entire process is beyond the scope of this book, we recommend installing Faiss using Anaconda or using its Docker containers to quickly get started.

Faiss

Faiss is Facebook's efficient similarity search library. It can scale to billions of vectors in RAM on a single server by storing a compressed representation of the vectors (compact quantization codes) instead of the original values. It's especially suited for dense vectors. It shines particularly on machines with GPUs by storing the index on GPU memory (VRAM). This works on both single-GPU and multi-GPU setups. It provides the ability to configure performance based on search time, accuracy, memory usage, and indexing time. It's one of the fastest known implementations of ANN search on GPU. Hey, if it's good enough for Facebook, it's good enough for most of us (as long as we have enough data).

While showing the entire process is beyond the scope of this book, we recommend installing Faiss using Anaconda or using its Docker containers to quickly get started.

Improving Accuracy with Fine Tuning

Many of the pretrained models were trained on the ImageNet dataset. Therefore, they provide an incredible starting point for similarity computations in most situations. That said, if you tuned these models to adapt to your specific problem, they would perform even more accurately at finding similar images.

In this portion of the chapter, we identify the worst-performing categories, visualize them with t-SNE, fine tune, and then see how their t-SNE graph changes.

What is a good metric to check whether you are indeed getting similar images?

Painful option 1

> Go through the entire dataset one image at a time, and manually score whether the returned images indeed look similar.

Happier option 2

> Simply calculate accuracy. That is, for an image belonging to category *X*, are the similar images belonging to the same category? We will refer to this similarity accuracy.

So, what are our worst-performing categories? And why are they the worst? To answer this, we have predefined a helper function `worst_classes`. For every image in the dataset, it finds the nearest neighbors using the brute-force algorithm and then returns six classes with the least accuracy. To see the effects of fine tuning, we run our analysis on a more difficult dataset: Caltech-256. Calling this function unveils the least-accurate classes:

```
names_of_worst_classes_before_finetuning, accuracy_per_class_before_finetuning =
worst_classes(feature_list[:])
```

```
Accuracy is 56.54
Top 6 incorrect classifications
059.drinking-straw    Accuracy: 11.76%
135.mailbox             Accuracy: 16.03%
108.hot-dog             Accuracy: 16.72%
163.playing-card      Accuracy: 17.29%
195.soda-can            Accuracy: 19.68%
```

```
125.knife          Accuracy: 20.53%
```

To see why they are performing so poorly on certain classes, we've plotted a t-SNE graph to visualize the embeddings in 2D space, which you can see in Figure 4-16. To prevent overcrowding on our plot, we use only 50 items from each of the 6 classes.

TIP

To enhance the visibility of the graph we can define different markers and different colors for each class. Matplotlib provides a wide variety of markers and colors.

```
markers = [ "^", ".","s", "o","x", "P" ]
colors = ['red', 'blue', 'fuchsia', 'green',
'purple', 'orange']
```

Figure 4-16. t-SNE visualization of feature vectors of least-accurate classes before fine tuning

Aah, these feature vectors are all over the place and on top of one another. Using these feature vectors in other applications such as classification might not be a good idea because it would be difficult to find a clean plane of separation between them. No wonder they performed so poorly in this nearest neighbor–based classification test.

What do you think will be the result if we repeat these steps with the fine-tuned model? We reckon something interesting; let's take a look at Figure 4-17 to see.

Figure 4-17. t-SNE visualization of feature vectors of least-accurate classes after fine tuning

This is so much cleaner. With just a little bit of fine tuning as shown in Chapter 3, the embeddings begin to group together. Compare the noisy/scattered embeddings of the pretrained models against those of the fine-tuned model. A machine learning classifier would be able to find a plane of separation between these classes with much more ease, hence yielding better classification accuracy as well as more similar images when not using a classifier. And, remember, these were the classes with the highest misclassifications; imagine how nicely the

classes with originally higher accuracy would be after fine tuning.

Previously, the pretrained embeddings achieved 56% accuracy. The new embeddings after fine tuning deliver a whopping 87% accuracy! A little magic goes a long way.

The one limitation for fine tuning is the requirement of labeled data, which is not always present. So depending on your use case, you might need to label some amount of data.

There’s a small unconventional training trick involved, though, which we discuss in the next section.

classes with originally higher accuracy would be after fine tuning.

Previously, the pretrained embeddings achieved 56% accuracy. The new embeddings after fine tuning deliver a whopping 87% accuracy! A little magic goes a long way.

The one limitation for fine tuning is the requirement of labeled data, which is not always present. So depending on your use case, you might need to label some amount of data.

There’s a small unconventional training trick involved, though, which we discuss in the next section.

Fine Tuning Without Fully Connected Layers

As we already know, a neural network comprises three parts:

- Convolutional layers, which end up generating the feature vectors
- Fully connected layers
- The final classifier layer

Fine tuning, as the name suggests, involves tweaking a neural network lightly to adapt to a new dataset. It usually involves stripping off the fully connected layers (top layers), substituting them with new ones, and then training this newly composed neural network using this dataset. Training in this manner will cause two things:

- The weights in all the newly added fully connected layers will be significantly affected.
- The weights in the convolutional layers will be only slightly changed.

The fully connected layers do a lot of the heavy lifting to get maximum classification accuracy. As a result, the majority of the network that generates the feature vectors will change insignificantly. Thus, the feature vectors, despite fine tuning, will show little change.

Our aim is for similar-looking objects to have closer feature vectors, which fine tuning as described earlier fails to accomplish. By forcing all of the task-specific learning to happen in the convolutional layers, we can see much better results. How do we achieve that? *By removing all of the fully connected layers and placing a classifier layer directly after the convolutional layers (which generate the feature vectors).* This model is optimized for similarity search rather than classification.

To compare the process of fine tuning a model optimized for classification tasks as opposed to similarity search, let's recall how we fine tuned our model in Chapter 3 for classification:

```
from tf.keras.applications.resnet50 import ResNet50
model = ResNet50(weights='imagenet', include_top=False,
```

```
input_shape = (224,224,3))
input = Input(shape=(224, 224, 3))
x = model(input)
x = GlobalAveragePooling2D()(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(NUM_CLASSES, activation='softmax')(x)
model_classification_optimized = Model(inputs=input, outputs=x)
```

And here's how we fine tune our model for similarity search. Note the missing hidden dense layer in the middle:

```
from tf.keras.applications.resnet50 import ResNet50
model = ResNet50(weights='imagenet', include_top=False,
input_shape = (224,224,3))
input = Input(shape=(224, 224, 3))
x = model(input)
x = GlobalAveragePooling2D()(x)
# No dense or dropout layers
x = Dense(NUM_CLASSES, activation='softmax')(x)
model_similarity_optimized = Model(inputs=input, outputs=x)
```

After fine tuning, to use the `model_similarity_optimized` for extracting features instead of giving probabilities for classes, simply `pop` (i.e., remove) the last layer:

```
model_similarity_optimized.layers.pop()
model = Model(model_similarity_optimized.input,
model_similarity_optimized.layers[-1].output)
```

The key thing to appreciate here is if you used the regular fine-tuning process, we would get lower similarity accuracy than `model_similarity_optimized`. Obviously, we would want to use `model_classification_optimized` for classification scenarios and `model_similarity_optimized` for extracting embeddings for similarity search.

With all this knowledge, you can now make both a fast and accurate similarity system for any scenario you are working on. It's time to see how the giants in the AI industry build their products.

Siamese Networks for One-Shot Face Verification

A face verification system is usually trying to ascertain—given two images of faces—whether the two images are of the same person. This is a high-precision binary classifier that needs to robustly work with different lighting, clothing, hairstyles, backgrounds, and facial expressions. To make things more challenging, although there might be images of many people in, for instance an employee database, there might be only a handful of images of the same person available. Similarly, signature identification in banks and product identification on Amazon suffer the same challenge of limited images per item.

How would you go about training such a classifier? Picking embeddings from a model like ResNet pretrained on ImageNet might not discern these fine facial attributes. One approach is to put each person as a separate class and then train like we usually train a regular network. Two key issues arise:

- If we had a million individuals, training for a million categories is not feasible.
- Training with a few images per class will lead to overtraining.

Another thought: instead of teaching different categories, we could teach a network to directly compare and decide whether a pair of images are similar or dissimilar by giving guidance on their similarity during training. And this is the key idea behind Siamese networks. Take a model, feed in two images, extract two embeddings, and then calculate the distance between the two embeddings. If the distance is under a threshold, consider them similar, else not. By feeding a pair of images with the associated label, similar or dissimilar, and training the network end to end, the embeddings begin to capture the fine-grained representation of the inputs. This approach, shown in Figure 4-18, of directly optimizing for the distance metric is called *metric learning*.

Figure 4-18. A Siamese network for signature verification; note that the same CNN was used for both inpu images

We could extend this idea and even feed three images. Pick one anchor image, pick another positive sample (of the same category), and another negative sample (of a different category). Let's now train this network to directly optimiz for the distance between similar items to be minimized and the distance between dissimilar items to be maximized. This loss function that helps us achieve this is called a *triplet loss* function. In the previous case with a pair of images, the loss function is called a *contrastive loss* function. The triplet loss function tends to give better results.

After the network is trained, we need only one reference image of a face for deciding at test time whether the person is the same. This methodology opens th doors for *one-shot learning*. Other common uses include signature and logo recognition. One remarkably creative application by Saket Maheshwary and Hemant Misra is to use a Siamese network for matching résumés with job applicants by calculating the semantic similarity between the two.

Figure 4-18. A Siamese network for signature verification; note that the same CNN was used for both input images

We could extend this idea and even feed three images. Pick one anchor image, pick another positive sample (of the same category), and another negative sample (of a different category). Let's now train this network to directly optimize for the distance between similar items to be minimized and the distance between dissimilar items to be maximized. This loss function that helps us achieve this is called a *triplet loss* function. In the previous case with a pair of images, the loss function is called a *contrastive loss* function. The triplet loss function tends to give better results.

After the network is trained, we need only one reference image of a face for deciding at test time whether the person is the same. This methodology opens the doors for *one-shot learning*. Other common uses include signature and logo recognition. One remarkably creative application by Saket Maheshwary and Hemant Misra is to use a Siamese network for matching résumés with job applicants by calculating the semantic similarity between the two.

Case Studies

Let’s look at a few interesting examples that show how what we have learned so far is applied in the industry.

Case Studies

Let’s look at a few interesting examples that show how what we have learned so far is applied in the industry.

Flickr

Flickr is one of the largest photo-sharing websites, especially popular among professional photographers. To help photographers find inspiration as well as showcase content the users might find interesting, Flickr produced a similarity search feature based on the same semantic meaning. As demonstrated in Figure 4-19, exploring a desert pattern leads to several similarly patterned results. Under the hood, Flickr adopted an ANN algorithm called Locally Optimized Product Quantization (LOPQ), which has been open sourced in Python as well as Spark implementations.

Figure 4-19. Similar patterns of a desert photo (image source)

Flickr

Flickr is one of the largest photo-sharing websites, especially popular among professional photographers. To help photographers find inspiration as well as showcase content the users might find interesting, Flickr produced a similarity search feature based on the same semantic meaning. As demonstrated in Figure 4-19, exploring a desert pattern leads to several similarly patterned results. Under the hood, Flickr adopted an ANN algorithm called Locally Optimized Product Quantization (LOPQ), which has been open sourced in Python as well as Spark implementations.

Figure 4-19. Similar patterns of a desert photo (image source)

Pinterest

Pinterest is an application used widely for its visual search capabilities, more specifically in its features called Similar Pins and Related Pins. Other companies like Baidu and Alibaba have launched similar visual search systems. Also, Zappos, Google Shopping, and like.com are using computer vision for recommendation.

Within Pinterest "women's fashion" is one of the most popular themes of pins and the Similar Looks feature (Figure 4-20) helps people discover similar products. Additionally, Pinterest also reports that its Related Pins feature increased its repin rate. Not every pin on Pinterest has associated metadata, which makes recommendation a difficult cold-start problem due to lack of context. Pinterest developers solved this cold-start problem by using the visual features for generating the related pins. Additionally, Pinterest implements an incremental fingerprinting service that generates new digital signatures if either a new image is uploaded or if there is feature evolution (due to improvements or modifications in the underlying models by the engineers).

Figure 4-20. The Similar Looks feature of the Pinterest application (image source: Pinterest blog)

Figure 4-20. The Similar Looks feature of the Pinterest application (image source: Pinterest blog)

Celebrity Doppelgangers

Website applications like *Celebslike.me*, which went viral in 2015, look for the nearest neighbor among celebrities, as shown in Figure 4-21. A similar viral approach was taken by the Google Arts & Culture app in 2018, which shows the nearest existing portrait to your face. Twins or not is another application with a similar aim.

Figure 4-21. Testing our friend Pete Warden's photo (technical lead for mobile and embedded TensorFlow at Google) on the celebslike.me website

Celebrity Doppelgangers

Website applications like *Celebslike.me*, which went viral in 2015, look for the nearest neighbor among celebrities, as shown in Figure 4-21. A similar viral approach was taken by the Google Arts & Culture app in 2018, which shows the nearest existing portrait to your face. Twins or not is another application with a similar aim.

Figure 4-21. Testing our friend Pete Warden's photo (technical lead for mobile and embedded TensorFlow at Google) on the celebslike.me website

Spotify

Spotify uses nearest neighbors for recommending music and creating automatic playlists and radio stations based on the current set of songs being played. Usually, collaborative filtering techniques, which are employed for recommending content like movies on Netflix, are content agnostic; that is, the recommendation happens because large groups of users with similar tastes are watching similar movies or listening to similar songs. This presents a problem for new and not yet popular content because users will keep getting recommendations for existing popular content. This is also referred to as the aforementioned cold-start problem. The solution is to use the latent understanding of the content. Similar to images, we can create feature vectors out of music using MFCC features (Mel Frequency Cepstral Coefficients), whicl in turn generates a 2D spectrogram that can be thought of as an image and can b used to generate features. Songs are divided into three-second fragments, and their spectrograms are used to generate features. These features are then averaged together to represent the complete song. Figure 4-22 shows artists whose songs are projected in specific areas. We can discern hip-hop (upper left), rock (upper right), pop (lower left), and electronic music (lower right). As already discussed, Spotify uses Annoy in the background.

Figure 4-22. t-SNE visualization of the distribution of predicted usage patterns, using latent factors predicted from audio (image source: "Deep content-based music recommendation" by Aaron van den Oord Sander Dieleman, Benjamin Schrauwen, NIPS 2013)

Figure 4-22. t-SNE visualization of the distribution of predicted usage patterns, using latent factors predicted from audio (image source: "Deep content-based music recommendation" by Aaron van den Oord, Sander Dieleman, Benjamin Schrauwen, NIPS 2013)

Image Captioning

Image captioning is the science of translating an image into a sentence (as illustrated in Figure 4-23). Going beyond just object tagging, this requires a deeper visual understanding of the entire image and relationships between objects. To train these models, an open source dataset called MS COCO was released in 2014, which consists of more than 300,000 images along with object categories, sentence descriptions, visual question-answer pairs, and object segmentations. It serves as a benchmark for a yearly competition to see progress in image captioning, object detection, and segmentation.

Figure 4-23. Image captioning feature in Seeing AI: the Talking Camera App for the blind community

A common strategy applied in the first year of the challenge (2015) was to append a language model (LSTM/RNN) with a CNN in such a way that the output of a CNN feature vector is taken as the input to the language model (LSTM/RNN). This combined model was trained jointly in an end-to-end manner, leading to very impressive results that stunned the world. Although every research lab was trying to beat one another, it was later found that doing a simple nearest-neighbor search could yield state-of-the-art results. For a given image, find similar images based on similarity of the embeddings. Then, note the common words in the captions of the similar images, and print the caption containing the most common words. In short, a lazy approach would still beat

the state-of-the-art one, and this exposed a critical bias in the dataset.

This bias has been coined the *Giraffe-Tree* problem by Larry Zitnick. Do an image search for “giraffe” on a search engine. Look closely: in addition to giraffe, is there grass in almost every image? Chances are you can describe the majority of these images as “A giraffe standing in a grass field.” Similarly, if a query image like the photo on the far left in Figure 4-24 contains a giraffe and a tree, almost all similar images (right) can be described as “a giraffe standing in the grass, next to a tree.” Even without a deeper understanding of the image, one would arrive at the correct caption using a simple nearest-neighbor search. This shows that to measure the real intelligence of a system, we need more semantically novel/original images in the test set.

Figure 4-24. The Giraffe-Tree problem (image source: Measuring Machine Intelligence Through Visual Question Answering, C. Lawrence Zitnick, Aishwarya Agrawal, Stanislaw Antol, Margaret Mitchell, Dhruv Batra, Devi Parikh)

In short, don’t underestimate a simple nearest-neighbor approach!

e

the state-of-the-art one, and this exposed a critical bias in the dataset.

This bias has been coined the *Giraffe-Tree* problem by Larry Zitnick. Do an image search for “giraffe” on a search engine. Look closely: in addition to giraffe, is there grass in almost every image? Chances are you can describe the majority of these images as “A giraffe standing in a grass field.” Similarly, if a query image like the photo on the far left in Figure 4-24 contains a giraffe and a tree, almost all similar images (right) can be described as “a giraffe standing in the grass, next to a tree.” Even without a deeper understanding of the image, one would arrive at the correct caption using a simple nearest-neighbor search. This shows that to measure the real intelligence of a system, we need more semantically novel/original images in the test set.

Figure 4-24. The Giraffe-Tree problem (image source: Measuring Machine Intelligence Through Visual Question Answering, C. Lawrence Zitnick, Aishwarya Agrawal, Stanislaw Antol, Margaret Mitchell, Dhruv Batra, Devi Parikh)

In short, don’t underestimate a simple nearest-neighbor approach!

Summary

Now we are at the end of a successful expedition where we explored locating similar images with the help of embeddings. We took this one level further by exploring how to scale searches from a few thousand to a few billion documents with the help of ANN algorithms and libraries including Annoy, NGT, and Faiss We also learned that fine tuning the model to your dataset can improve the accuracy and representative power of embeddings in a supervised setting. To top it all off, we looked at how to use Siamese networks, which use the power of embeddings to do one-shot learning, such as for face verification systems. We finally examined how nearest-neighbor approaches are used in various use cases across the industry. Nearest neighbors are a simple yet powerful tool to have in your toolkit.

Summary

Now we are at the end of a successful expedition where we explored locating similar images with the help of embeddings. We took this one level further by exploring how to scale searches from a few thousand to a few billion documents with the help of ANN algorithms and libraries including Annoy, NGT, and Faiss. We also learned that fine tuning the model to your dataset can improve the accuracy and representative power of embeddings in a supervised setting. To top it all off, we looked at how to use Siamese networks, which use the power of embeddings to do one-shot learning, such as for face verification systems. We finally examined how nearest-neighbor approaches are used in various use cases across the industry. Nearest neighbors are a simple yet powerful tool to have in your toolkit.

Chapter 5. From Novice to Master Predictor: Maximizing Convolutional Neural Network Accuracy

In Chapter 1, we looked at the importance of responsible AI development. One of the aspects we discussed was the importance of robustness of our models. Users can trust what we build only if they can be assured that the AI they encounter on a day-to-day basis is accurate and reliable. Obviously, the context of the application matters a lot. It would be okay for a food classifier to misclassify pasta as bread on occasion. But it would be dangerous for a self-driving car to misclassify a pedestrian as a street lane. The main goal of this chapter is thus a rather important one—to build more accurate models.

In this chapter, you will develop an intuition for recognizing opportunities to improve your model's accuracy the next time you begin training one. We first look at the tools that will ensure that you won't be going in blind. After that, for a good chunk of this chapter, we take a very experimental approach by setting up a baseline, isolating individual parameters to tweak, and observing their effect on model performance and training speed. A lot of the code we use in this chapter is all aggregated in a single Jupyter Notebook, along with an actionable checklist with interactive examples. It is meant to be highly reusable should you choose to incorporate it in your next training script.

We explore several questions that tend to come up during model training:

- I am unsure whether to use transfer learning or building from scratch to train my own network. What is the preferred approach for my scenario?
- What is the least amount of data that I can supply to my training pipeline to get acceptable results?
- I want to ensure that the model is learning the correct thing and not picking

up spurious correlations. How can I get visibility into that?

- How can I ensure that I (or someone else) will obtain the same results from my experiments every single time they are run? In other words, how do I ensure reproducibility of my experiments?
- Does changing the aspect ratio of the input images have an impact on the predictions?
- Does reducing input image size have a significant effect on prediction results?
- If I use transfer learning, what percentage of layers should I fine tune to achieve my preferred balance of training time versus accuracy?
- Alternatively, if I were to train from scratch, how many layers should I have in my model?
- What is the appropriate “learning rate” to supply during model training?
- There are too many things to remember. Is there a way to automate all of this work?

We will try to answer these questions one by one in the form of experiments on a few datasets. Ideally, you should be able to look at the results, read the takeaways, and gain some insight into the concept that the experiment was testing. If you’re feeling more adventurous, you can choose to perform the experiments yourself using the Jupyter Notebook.

up spurious correlations. How can I get visibility into that?

- How can I ensure that I (or someone else) will obtain the same results from my experiments every single time they are run? In other words, how do I ensure reproducibility of my experiments?
- Does changing the aspect ratio of the input images have an impact on the predictions?
- Does reducing input image size have a significant effect on prediction results?
- If I use transfer learning, what percentage of layers should I fine tune to achieve my preferred balance of training time versus accuracy?
- Alternatively, if I were to train from scratch, how many layers should I have in my model?
- What is the appropriate “learning rate” to supply during model training?
- There are too many things to remember. Is there a way to automate all of this work?

We will try to answer these questions one by one in the form of experiments on a few datasets. Ideally, you should be able to look at the results, read the takeaways, and gain some insight into the concept that the experiment was testing. If you’re feeling more adventurous, you can choose to perform the experiments yourself using the Jupyter Notebook.

Tools of the Trade

One of the main priorities of this chapter is to reduce the code and effort involved during experimentation while trying to gain insights into the process in order to reach high accuracy. An arsenal of tools exists that can assist us in making this journey more pleasant:

TensorFlow Datasets

Quick and easy access to around 100 datasets in a performant manner. All well-known datasets are available starting from the smallest MNIST (a few megabytes) to the largest MS COCO, ImageNet, and Open Images (several hundred gigabytes). Additionally, medical datasets like the Colorectal Histology and Diabetic Retinopathy are also available.

TensorBoard

Close to 20 easy-to-use methods to visualize many aspects of training, including visualizing the graph, tracking experiments, and inspecting the images, text, and audio data that pass through the network during training.

What-If Tool

Run experiments in parallel on separate models and tease out differences in them by comparing their performance on specific data points. Edit individual data points to see how that affects the model training.

tf-explain

Analyze decisions made by the network to identify bias and inaccuracies in the dataset. Additionally, use heatmaps to visualize what parts of the image the network activated on.

Keras Tuner

A library built for `tf.keras` that enables automatic tuning of hyperparameters in TensorFlow 2.0.

AutoKeras

Automates Neural Architecture Search (NAS) across different tasks like image, text, and audio classification and image detection.

AutoAugment

Utilizes reinforcement learning to improve the amount and diversity of data in an existing training dataset, thereby increasing accuracy.

Let’s now explore these tools in greater detail.

Automates Neural Architecture Search (NAS) across different tasks like image, text, and audio classification and image detection.

AutoAugment

Utilizes reinforcement learning to improve the amount and diversity of data in an existing training dataset, thereby increasing accuracy.

Let’s now explore these tools in greater detail.

TensorFlow Datasets

TensorFlow Datasets is a collection of nearly 100 ready-to-use datasets that can quickly help build high-performance input data pipelines for training TensorFlow models. Instead of downloading and manipulating data sets manually and then figuring out how to read their labels, TensorFlow Datasets standardizes the data format so that it's easy to swap one dataset with another, often with just a single line of code change. As you will see later on, doing things like breaking the dataset down into training, validation, and testing is also a matter of a single line of code. We will additionally be exploring TensorFlow Datasets from a performance point of view in the next chapter.

You can list all of the available datasets by using the following command (in the interest of conserving space, only a small subset of the full output is shown in this example):

```
import tensorflow_datasets as tfds
print(tfds.list_builders())
```

```
['amazon_us_reviews', 'bair_robot_pushing_small', 'bigearthnet', 'caltech101',
'cats_vs_dogs', 'celeb_a', 'imagenet2012', … , 'open_images_v4',
'oxford_flowers102', 'stanford_dogs','voc2007', 'wikipedia', 'wmt_translate',
'xnli']
```

Let's see how simple it is to load a dataset. We will plug this into a full working pipeline later:

```
# Import necessary packages
import tensorflow_datasets as tfds

# Downloading and loading the dataset
dataset = tfds.load(name="cats_vs_dogs", split=tfds.Split.TRAIN)

# Building a performance data pipeline
dataset = dataset.map(preprocess).cache().repeat().shuffle(1024).batch(32).
prefetch(tf.data.experimental.AUTOTUNE)

model.fit(dataset, ...)
```

TIP

`tfds` generates a lot of progress bars, and they take up a lot of screen space—using `tfds.disable_progress_bar()` might be a good idea.

TIP

`tfds` generates a lot of progress bars, and they take up a lot of screen space—using `tfds.disable_progress_bar()` might be a good idea.

TensorBoard

TensorBoard is a one-stop-shop for all of your visualization needs, offering close to 20 tools to understand, inspect, and improve your model's training.

Traditionally, to track experiment progress, we save the values of loss and accuracy per epoch and then, when done, plot it using `matplotlib`. The downside with that approach is that it's not real time. Our usual options are to watch for the training progress in text. Additionally, after the training is done, we need to write additional code to make the graph in `matplotlib`. TensorBoard solves these and more pressing issues by offering a real-time dashboard (Figure 5-1) that helps us visualize all logs (such as train/validation accuracy and loss) to assist in understanding the progression of training. Another benefit it offers is the ability to compare our current experiment's progress with the previous experiment, so we can see how a change in parameters affected our overall accuracy.

Figure 5-1. TensorBoard default view showcasing real-time training metrics (the lightly shaded lines represent the accuracy from the previous run)

To enable TensorBoard to visualize our training and models, we need to log information about our training with the help of summary writer:

```
summary_writer = tf.summary.FileWriter('./logs')
```

To follow our training in real time, we need to load TensorBoard before the model training begins. We can load TensorBoard by using the following commands:

```
# Get TensorBoard to run
%load_ext tensorboard

# Start TensorBoard
%tensorboard --logdir ./log
```

As more TensorFlow components need a visual user interface, they reuse TensorBoard by becoming embeddable plug-ins within it. You'll notice the Inactive drop-down menu on TensorBoard; that's where you can see all the different profiles or tools that TensorFlow offers. Table 5-1 showcases a handful of the wide variety of tools available.

Table 5-1. Plugins for TensorBoard

TensorBoard plug-in name	Description
Default Scalar	Visualize scalar values such as classification accuracy.
Custom Scalar	Visualize user-defined custom metrics. For example, different weights for different classes, which might not be a readily available metric.
Image	View the output from each layer by clicking the Images tab.
Audio	Visualize audio data.
Debugging tools	Allows debugging visually and setting conditional breakpoints (e.g., tensor contains Nan or Infinity).
Graphs	Shows the model architecture graphically.
Histograms	Show the changes in the weight distribution in the layers of a model as the training progresses. This is especially useful for checking the effect of compressing a model with quantization.
Projector	Visualize projections using t-SNE, PCA, and others.
Text	Visualize text data.
PR curves	Plot precision-recall curves.
Profile	Benchmark speed of all operations and layers in a model.
Beholder	Visualize the gradients and activations of a model in real time during training. It allows seeing them filter by filter, and allows them to be exported as images or even as a video.
What-If Tool	For investigating the model by slicing and dicing the data and checking its

	performance. Especially helpful for discovering bias.
HParams	Find out which params and at what values are the most important, allow logging of the entire parameter server (discussed in detail in this chapter).
Mesh	Visualize 3D data (including point clouds).

It should be noted that TensorBoard is not TensorFlow specific, and can be used with other frameworks like PyTorch, scikit-learn, and more, depending on the plugin used. To make a plugin work, we need to write the specific metadata that we want to visualize. For example, TensorBoard embeds the TensorFlow Projector tool within to cluster images, text, or audio using t-SNE (which we examined in detail in Chapter 4). Apart from calling TensorBoard, we need to write the metadata like the feature embeddings of our image, so that TensorFlow Projector can use it to do clustering, as demonstrated in Figure 5-2.

Figure 5-2. TensorFlow Embedding Projector showcasing data in clusters (can be run as a TensorBoard plugin)

	performance. Especially helpful for discovering bias.
HParams	Find out which params and at what values are the most important, allow logging of the entire parameter server (discussed in detail in this chapter).
Mesh	Visualize 3D data (including point clouds).

It should be noted that TensorBoard is not TensorFlow specific, and can be used with other frameworks like PyTorch, scikit-learn, and more, depending on the plugin used. To make a plugin work, we need to write the specific metadata that we want to visualize. For example, TensorBoard embeds the TensorFlow Projector tool within to cluster images, text, or audio using t-SNE (which we examined in detail in Chapter 4). Apart from calling TensorBoard, we need to write the metadata like the feature embeddings of our image, so that TensorFlow Projector can use it to do clustering, as demonstrated in Figure 5-2.

Figure 5-2. TensorFlow Embedding Projector showcasing data in clusters (can be run as a TensorBoard plugin)

What-If Tool

What if we could inspect our AI model's predictions with the help of visualizations? What if we could find the best threshold for our model to maximize precision and recall? What if we could slice and dice the data along with the predictions our model made to see what it's great at and where there are opportunities to improve? What if we could compare two models to figure out which is indeed better? What if we could do all this and more, with a few clicks in the browser? Sounds appealing for sure! The What-If Tool (Figure 5-3 and Figure 5-4) from Google's People + AI Research (PAIR) initiative helps open up the black box of AI models to enable model and data explainability.

Figure 5-3. What-If Tool's datapoint editor makes it possible to filter and visualize data according to annotations of the dataset and labels from the classifier

Figure 5-4. PR curves in the Performance and Fairness section of the What-If Tool help to interactively select the optimal threshold to maximize precision and recall

To use the What-If Tool, we need the dataset and a model. As we just saw, TensorFlow Datasets makes downloading and loading the data (in the `tfrecord` format) relatively easy. All we need to do is to locate the data file. Additionally, we want to save the model in the same directory:

```
# Save model for What If Tool
tf.saved_model.save(model, "/tmp/model/1/")
```

It's best to perform the following lines of code in a local system rather than a Colab notebook because the integration between Colab and the What-If Tool is still evolving.

Let's start TensorBoard:

```
$ mkdir tensorboard
$ tensorboard --logdir ./log --alsologtostderr
```

Now, in a new terminal, let's make a directory for all of our What-If Tool experiments:

```
$ mkdir what-if-stuff
```

Move the trained model and TFRecord data here. The overall directory structure looks something like this:

```
$ tree .
├── colo
|   └── model
|         └── 1
|         ├── assets
|         ├── saved_model.pb
|         └── variables
```

We'll serve the model using Docker within the newly created directory:

```
$ sudo docker run -p 8500:8500 \
--mount type=bind,source=/home/{your_username}/what-if-stuff/colo/model/,
 target=/models/colo \
-e MODEL_NAME=colo -t tensorflow/serving
```

A word of caution: the port must be `8500` and all parameters must be spelled exactly as shown in the preceding example.

Next, at the far right, click the settings button (the gray gear icon) and add the values listed in Table 5-2.

Table 5-2. Configurations for the What-If Tool

Parameter	Value
Inference address	`ip_addr:8500`
Model name	`/models/colo`
Model type	Classification
Path to examples	*/home/{your_username}/what_if_stuff/colo/models/colo.tfrec* (Note: this must be an absolute path)

We can now open the What-If Tool in the browser within TensorBoard, as depicted in Figure 5-5.

Figure 5-5. Setup window for the What-If Tool

The What-If Tool can also be used to visualize datasets according to different bins, as shown in Figure 5-6. We can also use the tool to determine the better performing model out of multiple models on the same dataset using the `set_compare_estimator_and_feature_spec` function.

```
from witwidget.notebook.visualization import WitConfigBuilder

# features are the test examples that we want to load into the tool
models = [model2, model3, model4]
config_builder =
WitConfigBuilder(test_examples).set_estimator_and_feature_spec(model1, features)

for each_model in models:
    config_builder =
 config_builder.set_compare_estimator_and_feature_spec(each_model, features)
```

Figure 5-6. The What-If tool enables using multiple metrics, data visualization, and many more things under the sun

Now, we can load TensorBoard, and then, in the Visualize section, choose the model we want to compare, as shown in Figure 5-7. This tool has many features to explore!

Figure 5-7. Choose the model to compare using the What-If Tool

Figure 5-6. The What-If tool enables using multiple metrics, data visualization, and many more things under the sun

Now, we can load TensorBoard, and then, in the Visualize section, choose the model we want to compare, as shown in Figure 5-7. This tool has many features to explore!

Figure 5-7. Choose the model to compare using the What-If Tool

tf-explain

Deep learning models have traditionally been black boxes, and up until now, we usually learn about their performance by watching the class probabilities and validation accuracies. To make these models more interpretable and explainable, heatmaps come to the rescue. By showing the area of an image that leads to the prediction with higher intensity, heatmaps can help visualize their learning. For example, an animal often seen in surroundings with snow might be getting high-accuracy predictions, but if the dataset has only that animal with snow as the background, the model might just be paying attention to the snow as the distinctive pattern instead of the animal. Such a dataset demonstrates bias, making the predictions not too robust when the classifier is put in the real world (and potentially dangerous!). Heatmaps can be especially useful to explore such bias, as often spurious correlations can seep in if the dataset is not carefully curated.

`tf-explain` (by Raphael Meudec) helps understand the results and inner workings of a neural network with the help of such visualizations, removing the veil on bias in datasets. We can add multiple types of callbacks while training or use its core API to generate TensorFlow events that can later be loaded into TensorBoard. For inference, all we need to do is pass an image, its ImageNet object ID along with a model into tf-explain's functions. You must supply the object ID because `tf.explain` needs to know what is activated for that particular class. A few different visualization approaches are available with `tf.explain`:

Grad CAM

> The Gradient-weighted Class Activation Mapping (Grad CAM) visualizes how parts of the image affect the neural network's output by looking into the activation maps. A heatmap (illustrated in Figure 5-8) is generated based on the gradients of the object ID from the last convolutional layer. Grad CAM is largely a broad-spectrum heatmap generator given that it is robust to noise and can be used on an array of CNN models.

Occlusion Sensitivity

Occludes a part of the image (using a small square patch placed randomly) to establish how robust the network is. If the prediction is still correct, on average, the network is robust. The area in the image that is the warmest (i.e., red) has the most effect on the prediction when occluded.

Activations

Visualizes the activations for the convolutional layers.

Figure 5-8. Visualizations on images using MobileNet and tf-explain

As demonstrated in the code example that follows, such visualizations can be built with very little code. By taking a video, generating individual frames, and running tf-explain with Grad CAM and joining them together, we can build a detailed understanding of how these neural networks would react to moving camera angles.

```
from tf_explain.core.grad_cam import GradCAM
From tf.keras.applications.MobileNet import MobileNet

model = MobileNet(weights='imagenet', include_top=True)

# Set Grad CAM System
explainer = GradCAM()

# Image Processing
IMAGE_PATH = 'dog.jpg'
dog_index = 263
```

```
img = tf.keras.preprocessing.image.load_img(IMAGE_PATH, target_size=(224, 224))
img = tf.keras.preprocessing.image.img_to_array(img)
data = ([img], None)

# Passing the image through Grad CAM
grid = explainer.explain(data, model, 'conv1', index)
name = IMAGE_PATH.split(".jpg")[0]
explainer.save(grid, '/tmp', name + '_grad_cam.png')
```

```
img = tf.keras.preprocessing.image.load_img(IMAGE_PATH, target_size=(224, 224))
img = tf.keras.preprocessing.image.img_to_array(img)
data = ([img], None)

# Passing the image through Grad CAM
grid = explainer.explain(data, model, 'conv1', index)
name = IMAGE_PATH.split(".jpg")[0]
explainer.save(grid, '/tmp', name + '_grad_cam.png')
```

Common Techniques for Machine Learning Experimentation

The first few chapters focused on training the model. The following sections, however, contain a few more things to keep in the back of your mind while running your training experiments.

Common Techniques for Machine Learning Experimentation

The first few chapters focused on training the model. The following sections, however, contain a few more things to keep in the back of your mind while running your training experiments.

Data Inspection

Data inspection's first biggest hurdle is determining the structure of the data. TensorFlow Datasets has made this step relatively easy because all of the available datasets are in the same format and structure and can be used in a performant way. All we need to do is load the dataset into the What-If Tool and use the various options already present to inspect the data. As an example, on the SMILE dataset, we can visualize the dataset according to its annotations, such as images of people wearing eyeglasses and those without eyeglasses, as illustrated in Figure 5-9. We observe that a wider distribution of the dataset has images of people wearing no eyeglasses, thus uncovering bias in the data due to an unbalanced dataset. This can be solved by modifying the weights of the metrics accordingly, through the tool.

Figure 5-9. Slicing and dividing the data based on predictions and real categories

Data Inspection

Data inspection's first biggest hurdle is determining the structure of the data. TensorFlow Datasets has made this step relatively easy because all of the available datasets are in the same format and structure and can be used in a performant way. All we need to do is load the dataset into the What-If Tool and use the various options already present to inspect the data. As an example, on the SMILE dataset, we can visualize the dataset according to its annotations, such as images of people wearing eyeglasses and those without eyeglasses, as illustrated in Figure 5-9. We observe that a wider distribution of the dataset has images of people wearing no eyeglasses, thus uncovering bias in the data due to an unbalanced dataset. This can be solved by modifying the weights of the metrics accordingly, through the tool.

Figure 5-9. Slicing and dividing the data based on predictions and real categories

Breaking the Data: Train, Validation, Test

Splitting a dataset into train, validation, and test is pretty important because we want to report the results on an unseen dataset by the classifier (i.e., the test dataset). TensorFlow Datasets makes it easy to download, load, and split the dataset into these three parts. Some datasets already come with three default splits. Alternatively, the data can be split by percentages. The following code showcases using a default split:

```
dataset_name = "cats_vs_dogs"
train, info_train = tfds.load(dataset_name, split=tfds.Split.TRAIN,
                    with_info=True)
```

The cats-and-dogs dataset in `tfds` has only the train split predefined. Similar to this, some datasets in TensorFlow Datasets do not have a `validation` split. For those datasets, we take a small percentage of samples from the predefined `training` set and treat it as the `validation` set. To top it all off, splitting the dataset using the `weighted_splits` takes care of randomizing and shuffling data between the splits:

```
# Load the dataset
dataset_name = "cats_vs_dogs"

# Dividing data into train (80), val (10) and test (10)
split_train, split_val, split_test = tfds.Split.TRAIN.subsplit(weighted=[80, 10,
                                     10])
train, info_train = tfds.load(dataset_name, split=split_train , with_info=True)
val, info_val = tfds.load(dataset_name, split=split_val, with_info=True)
test, info_test = tfds.load(dataset_name, split=split_test, with_info=True)
```

Breaking the Data: Train, Validation, Test

Splitting a dataset into train, validation, and test is pretty important because we want to report the results on an unseen dataset by the classifier (i.e., the test dataset). TensorFlow Datasets makes it easy to download, load, and split the dataset into these three parts. Some datasets already come with three default splits. Alternatively, the data can be split by percentages. The following code showcases using a default split:

```
dataset_name = "cats_vs_dogs"
train, info_train = tfds.load(dataset_name, split=tfds.Split.TRAIN,
                    with_info=True)
```

The cats-and-dogs dataset in `tfds` has only the train split predefined. Similar to this, some datasets in TensorFlow Datasets do not have a `validation` split. For those datasets, we take a small percentage of samples from the predefined `training` set and treat it as the `validation` set. To top it all off, splitting the dataset using the `weighted_splits` takes care of randomizing and shuffling data between the splits:

```
# Load the dataset
dataset_name = "cats_vs_dogs"

# Dividing data into train (80), val (10) and test (10)
split_train, split_val, split_test = tfds.Split.TRAIN.subsplit(weighted=[80, 10,
                                     10])
train, info_train = tfds.load(dataset_name, split=split_train , with_info=True)
val, info_val = tfds.load(dataset_name, split=split_val, with_info=True)
test, info_test = tfds.load(dataset_name, split=split_test, with_info=True)
```

Early Stopping

Early stopping helps to avoid overtraining of the network by keeping a lookout for the number of epochs that show limited improvement. Assuming a model is set to train for 1,000 epochs and reaches 90% accuracy at the 10th epoch and stops improving any further for the next 10 epochs, it might be a waste of resources to train any further. If the number of epochs exceeds a predefined threshold called `patience`, training is stopped even if there might still be more epochs left to train. In other words, early stopping decides the point at which the training would no longer be useful and stops training. We can change the metric using the `monitor` parameter and add early stopping to our list of callbacks for the model:

```
# Define Early Stopping callback
earlystop_callback = tf.keras.callbacks.EarlyStopping(monitor='val_acc',
                                        min_delta=0.0001, patience=10)

# Add to the training model
model.fit_generator(... callbacks=[earlystop_callback])
```

Early Stopping

Early stopping helps to avoid overtraining of the network by keeping a lookout for the number of epochs that show limited improvement. Assuming a model is set to train for 1,000 epochs and reaches 90% accuracy at the 10th epoch and stops improving any further for the next 10 epochs, it might be a waste of resources to train any further. If the number of epochs exceeds a predefined threshold called `patience`, training is stopped even if there might still be more epochs left to train. In other words, early stopping decides the point at which the training would no longer be useful and stops training. We can change the metric using the `monitor` parameter and add early stopping to our list of callbacks for the model:

```
# Define Early Stopping callback
earlystop_callback = tf.keras.callbacks.EarlyStopping(monitor='val_acc',
                                        min_delta=0.0001, patience=10)

# Add to the training model
model.fit_generator(... callbacks=[earlystop_callback])
```

Reproducible Experiments

Train a network once. Then, train it again, without changing any code or parameters. You might notice that the accuracies in two subsequent runs came out slightly different, even if no change was made in code. This is due to random variables. To make experiments reproducible across runs, we want to control this randomization. Initialization of weights of models, randomized shuffling of data and so on all utilize randomization algorithms. We know that random number generators can be made reproducible by initializing a seed and that's exactly what we will do. Various frameworks have their own ways of setting a random seed, some of which are shown here:

```
# Seed for Tensorflow
tf.random.set_seed(1234)

# Seed for Numpy
import numpy as np
np.random.seed(1234)

# Seed for Keras
seed = 1234
fit(train_data, augment=True, seed=seed)
flow_from_dataframe(train_dataframe, shuffle=True, seed=seed)
```

NOTE

It is necessary to set a seed in all the frameworks and subframeworks that are being used, as seeds are not transferable between frameworks.

Reproducible Experiments

Train a network once. Then, train it again, without changing any code or parameters. You might notice that the accuracies in two subsequent runs came out slightly different, even if no change was made in code. This is due to random variables. To make experiments reproducible across runs, we want to control this randomization. Initialization of weights of models, randomized shuffling of data, and so on all utilize randomization algorithms. We know that random number generators can be made reproducible by initializing a seed and that's exactly what we will do. Various frameworks have their own ways of setting a random seed, some of which are shown here:

```
# Seed for Tensorflow
tf.random.set_seed(1234)

# Seed for Numpy
import numpy as np
np.random.seed(1234)

# Seed for Keras
seed = 1234
fit(train_data, augment=True, seed=seed)
flow_from_dataframe(train_dataframe, shuffle=True, seed=seed)
```

NOTE

It is necessary to set a seed in all the frameworks and subframeworks that are being used, as seeds are not transferable between frameworks.

End-to-End Deep Learning Example Pipeline

Let's combine several tools and build a skeletal backbone, which will serve as our pipeline in which we will add and remove parameters, layers, functionality, and various other addons to really understand what is happening. Following the code on the book's GitHub website (see *http://PracticalDeepLearning.ai*), you can interactively run this code for more than 100 datasets in your browser with Colab. Additionally, you can modify it for most classification tasks.

End-to-End Deep Learning Example Pipeline

Let's combine several tools and build a skeletal backbone, which will serve as our pipeline in which we will add and remove parameters, layers, functionality, and various other addons to really understand what is happening. Following the code on the book's GitHub website (see *http://PracticalDeepLearning.ai*), you can interactively run this code for more than 100 datasets in your browser with Colab. Additionally, you can modify it for most classification tasks.

Basic Transfer Learning Pipeline

First, let's build this end-to-end example for transfer learning.

```
# Import necessary packages
import tensorflow as tf
import tensorflow_datasets as tfds

# tfds makes a lot of progress bars, which takes up a lot of screen space, hence
# disabling them
tfds.disable_progress_bar()

tf.random.set_seed(1234)

# Variables
BATCH_SIZE = 32
NUM_EPOCHS= 20
IMG_H = IMG_W = 224
IMG_SIZE = 224
LOG_DIR = './log'
SHUFFLE_BUFFER_SIZE = 1024
IMG_CHANNELS = 3

dataset_name = "oxford_flowers102"

def preprocess(ds):
  x = tf.image.resize_with_pad(ds['image'], IMG_SIZE, IMG_SIZE)
  x = tf.cast(x, tf.float32)
  x = (x/127.5) - 1
  return x, ds['label']

def augmentation(image,label):
  image = tf.image.random_brightness(image, .1)
  image = tf.image.random_contrast(image, lower=0.0, upper=1.0)
  image = tf.image.random_flip_left_right(image)
  return image, label

def get_dataset(dataset_name):
  split_train, split_val = tfds.Split.TRAIN.subsplit(weighted=[9,1])
  train, info_train = tfds.load(dataset_name, split=split_train , with_info=True)
  val, info_val = tfds.load(dataset_name, split=split_val, with_info=True)
  NUM_CLASSES = info_train.features['label'].num_classes
  assert NUM_CLASSES >= info_val.features['label'].num_classes
  NUM_EXAMPLES = info_train.splits['train'].num_examples * 0.9
  IMG_H, IMG_W, IMG_CHANNELS = info_train.features['image'].shape
  train = train.map(preprocess).cache().
          repeat().shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
```

```
  train = train.map(augmentation)
  train = train.prefetch(tf.data.experimental.AUTOTUNE)
  val = val.map(preprocess).cache().repeat().batch(BATCH_SIZE)
  val = val.prefetch(tf.data.experimental.AUTOTUNE)
  return train, info_train, val, info_val, IMG_H, IMG_W, IMG_CHANNELS,
         NUM_CLASSES, NUM_EXAMPLES

train, info_train, val, info_val, IMG_H, IMG_W, IMG_CHANNELS, NUM_CLASSES,
NUM_EXAMPLES = get_dataset(dataset_name)

# Allow TensorBoard callbacks
tensorboard_callback = tf.keras.callbacks.TensorBoard(LOG_DIR,
                                                      histogram_freq=1,
                                                      write_graph=True,
                                                      write_grads=True,
                                                      batch_size=BATCH_SIZE,
                                                      write_images=True)


def transfer_learn(train, val, unfreeze_percentage, learning_rate):
   mobile_net = tf.keras.applications.ResNet50(input_shape=(IMG_SIZE, IMG_SIZE,
                IMG_CHANNELS), include_top=False)
   mobile_net.trainable=False
   # Unfreeze some of the layers according to the dataset being used
   num_layers = len(mobile_net.layers)
   for layer_index in range(int(num_layers - unfreeze_percentage*num_layers),
                            num_layers ):
                mobile_net.layers[layer_index].trainable = True
   model_with_transfer_learning = tf.keras.Sequential([mobile_net,
                         tf.keras.layers.GlobalAveragePooling2D(),
                         tf.keras.layers.Flatten(),
                         tf.keras.layers.Dense(64),
                         tf.keras.layers.Dropout(0.3),
                         tf.keras.layers.Dense(NUM_CLASSES,
                                               activation='softmax')],)
  model_with_transfer_learning.compile(
              optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=["accuracy"])
  model_with_transfer_learning.summary()
  earlystop_callback = tf.keras.callbacks.EarlyStopping(
                                  monitor='val_accuracy',
                                  min_delta=0.0001,
                                  patience=5)
  model_with_transfer_learning.fit(train,
                                  epochs=NUM_EPOCHS,
                                  steps_per_epoch=int(NUM_EXAMPLES/BATCH_SIZE),
                                  validation_data=val,
                                  validation_steps=1,
```

```
                                    validation_freq=1,
                                    callbacks=[tensorboard_callback,
                                               earlystop_callback])
  return model_with_transfer_learning

# Start TensorBoard
%tensorboard --logdir ./log

# Select the last % layers to be trained while using the transfer learning
# technique. These layers are the closest to the output layers.
unfreeze_percentage = .33
learning_rate = 0.001

model = transfer_learn(train, val, unfreeze_percentage, learning_rate)
```

```
                                    validation_freq=1,
                                    callbacks=[tensorboard_callback,
                                               earlystop_callback])
  return model_with_transfer_learning

# Start TensorBoard
%tensorboard --logdir ./log

# Select the last % layers to be trained while using the transfer learning
# technique. These layers are the closest to the output layers.
unfreeze_percentage = .33
learning_rate = 0.001

model = transfer_learn(train, val, unfreeze_percentage, learning_rate)
```

Basic Custom Network Pipeline

Apart from transfer learning on state-of-the-art models, we can also experiment and develop better intuitions by building our own custom network. Only the model needs to be swapped in the previously defined transfer learning code:

```
def create_model():
  model = tf.keras.Sequential([
     tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
          input_shape=(IMG_SIZE, IMG_SIZE, IMG_CHANNELS)),
     tf.keras.layers.MaxPool2D(pool_size=(2, 2)),
     tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),
     tf.keras.layers.MaxPool2D(pool_size=(2, 2)),
     tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),
     tf.keras.layers.MaxPool2D(pool_size=(2, 2)),
     tf.keras.layers.Dropout(rate=0.3),
     tf.keras.layers.Flatten(),
     tf.keras.layers.Dense(128, activation='relu'),
     tf.keras.layers.Dropout(rate=0.3),
     tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
  ])
  return model

def scratch(train, val, learning_rate):
  model = create_model()
  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])

  earlystop_callback = tf.keras.callbacks.EarlyStopping(
                monitor='val_accuracy',
                min_delta=0.0001,
                patience=5)

  model.fit(train,
            epochs=NUM_EPOCHS,
            steps_per_epoch=int(NUM_EXAMPLES/BATCH_SIZE),
            validation_data=val,
            validation_steps=1,
            validation_freq=1,
            callbacks=[tensorboard_callback, earlystop_callback])
  return model
```

Now, it’s time to use our pipeline for various experiments.

How Hyperparameters Affect Accuracy

In this section, we aim to modify various parameters of a deep learning pipeline one at a time—from the number of layers fine-tuned, to the choice of the activation function used—and see its effect primarily on validation accuracy. Additionally, when relevant, we also observe its effect on the speed of training and time to reach the best accuracy (i.e., convergence).

Our experimentation setup is as follows:

- To reduce experimentation time, we have used a faster architecture—MobileNet—in this chapter.
- We reduced the input image resolution to 128 x 128 pixels to further speed up training. In general, we would recommend using a higher resolution (at least 224 x 224) for production systems.
- Early stopping is applied to stop experiments if they don't increase in accuracy for 10 consecutive epochs.
- For training with transfer learning, we generally unfreeze the last 33% of the layers.
- Learning rate is set to 0.001 with Adam optimizer.
- We're mostly using the Oxford Flowers 102 dataset for testing, unless otherwise stated. We chose this dataset because it is reasonably difficult to train on due to the large number of classes it contains (102) and the similarities between many of the classes that force networks to develop a fine-grained understanding of features in order to do well.
- To make apples-to-apples comparisons, we take the maximum accuracy value in a particular experiment and normalize all other accuracy values within that experiment with respect to this maximum value.

Based on these and other experiments, we have compiled a checklist of actionable tips to implement in your next model training adventure. These are available on the book's GitHub (see *http://PracticalDeepLearning.ai*) along with

interactive visualizations. If you have more tips, feel free to tweet them @PracticalDLBook or submit a pull request.

interactive visualizations. If you have more tips, feel free to tweet them @PracticalDLBook or submit a pull request.

Transfer Learning Versus Training from Scratch

Experimental setup

Train two models: one using transfer learning, and one from scratch on the same dataset.

Datasets used

Oxford Flowers 102, Colorectal Histology

Architectures used

Pretrained MobileNet, Custom model

Figure 5-10 shows the results.

Figure 5-10. Comparing transfer learning versus training a custom model on different datasets

Here are the key takeaways:

- Transfer learning leads to a quicker rise in accuracy during training by reusing previously learned features.
- Although it is expected that transfer learning (based on pretrained models on ImageNet) would work when the target dataset is also of natural imagery, the patterns learned in the early layers by a network work surprisingly well for datasets beyond ImageNet. That does not necessarily mean that it will yield the best results, but it can get close. When the images match more real-world images that the model was pretrained on, we get relatively quick improvement in accuracy.

5

Effect of Number of Layers Fine-Tuned in Transfer Learning

Experimental setup

Vary the percentage of trainable layers from 0 to 100%

Dataset used

Oxford Flowers 102

Architecture used

Pretrained MobileNet

Figure 5-11 shows the results.

Figure 5-11. Effect of % layers fine-tuned on model accuracy

Here are the key takeaways:

- The higher the number of layers fine-tuned, the fewer epochs it took to reach convergence and the higher the accuracy.
- The higher the number of layers fine-tuned, the more time it took per epoch for training, due to more computation and updates involved.
- For a dataset that required fine-grained understanding of images, making more layers task specific by unfreezing them was the key to a better model.

Effect of Data Size on Transfer Learning

Experimental setup

Add one image per class at a time

Dataset used

Cats versus dogs

Architecture used

Pretrained MobileNet

Figure 5-12 shows the results.

Figure 5-12. Effect of the amount of data per category on model accuracy

Here are the key takeaways:

- Even with only three images in each class, the model was able to predict with close to 90% accuracy. This shows how powerful transfer learning can be in reducing data requirements.
- Because ImageNet has several cats and dogs, pretrained networks on ImageNet suited our dataset much more easily. More difficult datasets like Oxford Flowers 102 might require a much higher number of images to achieve similar accuracies.

Effect of Learning Rate

Experimental setup

Vary the learning rate between .1, .01, .001, and .0001

Dataset used

Oxford Flowers 102

Architecture used

Pretrained MobileNet

Figure 5-13 shows the results.

Figure 5-13. Effect of learning rate on model accuracy and speed of convergence

Here are the key takeaways:

- Too high of a learning rate, and the model might never converge.
- Too low a learning rate results in a long time taken to convergence.
- Striking the right balance is crucial in training quickly.

Effect of Optimizers

Experimental setup

Experiment with available optimizers including AdaDelta, AdaGrad, Adam, Gradient Descent, Momentum, and RMSProp

Dataset used

Oxford Flowers 102

Architecture used

Pretrained MobileNet

Figure 5-14 shows the results.

Figure 5-14. Effect of different optimizers on the speed of convergence

Here are the key takeaways:

- Adam is a great choice for faster convergence to high accuracy.
- RMSProp is usually better for RNN tasks.

Effect of Optimizers

Experimental setup

Experiment with available optimizers including AdaDelta, AdaGrad, Adam, Gradient Descent, Momentum, and RMSProp

Dataset used

Oxford Flowers 102

Architecture used

Pretrained MobileNet

Figure 5-14 shows the results.

Figure 5-14. Effect of different optimizers on the speed of convergence

Here are the key takeaways:

- Adam is a great choice for faster convergence to high accuracy.
- RMSProp is usually better for RNN tasks.

Effect of Batch Size

Experimental setup

Vary batch sizes in powers of two

Dataset used

Oxford Flowers 102

Architecture used

Pretrained

Figure 5-15 shows the results.

Figure 5-15. Effect of batch size on accuracy and speed of convergence

Here are the key takeaways:

- The higher the batch size, the more the instability in results from epoch to epoch, with bigger rises and drops. But higher accuracy also leads to more efficient GPU utilization, so faster speed per epoch.
- Too low a batch size slows the rise in accuracy.
- 16/32/64 are good to start batch sizes with.

Effect of Resizing

Experimental setup

Change image size to 128x128, 224x224

Dataset used

Oxford Flowers 102

Architecture used

Pretrained

Figure 5-16 shows the results.

Figure 5-16. Effect of image size on accuracy

Here are the key takeaways:

- Even with a third of the pixels, there wasn't a significant difference in validation accuracies. On the one hand, this shows the robustness of CNNs. It might partly be because the Oxford Flowers 102 dataset has close-ups of flowers visible. For datasets in which the objects have much smaller portions in an image, the results might be lower.

Effect of Resizing

Experimental setup

Change image size to 128x128, 224x224

Dataset used

Oxford Flowers 102

Architecture used

Pretrained

Figure 5-16 shows the results.

Figure 5-16. Effect of image size on accuracy

Here are the key takeaways:

- Even with a third of the pixels, there wasn't a significant difference in validation accuracies. On the one hand, this shows the robustness of CNNs. It might partly be because the Oxford Flowers 102 dataset has close-ups of flowers visible. For datasets in which the objects have much smaller portions in an image, the results might be lower.

Effect of Change in Aspect Ratio on Transfer Learning

Experimental Setup

Take images of various aspect ratios (width:height ratio) and resize them to a square (1:1 aspect ratio).

Dataset Used

Cats vs. Dogs

Architecture Used

Pretrained

Figure 5-17 shows the results.

Figure 5-17. Distribution of aspect ratio and corresponding accuracies in images

Here are the key takeaways:

- Most common aspect ratio is 4:3; that is, 1.33, whereas our neural networks are generally trained at 1:1 ratio.
- Neural networks are relatively robust to minor modifications in aspect ratio brought upon by resizing to a square shape. Even up to 2.0 ratio gives decent results.

Tools to Automate Tuning for Maximum Accuracy

As we have seen since the rise of the nineteenth century, automation has always led to an increase in productivity. In this section, we investigate tools that can help us automate the search for the best model.

Tools to Automate Tuning for Maximum Accuracy

As we have seen since the rise of the nineteenth century, automation has always led to an increase in productivity. In this section, we investigate tools that can help us automate the search for the best model.

Keras Tuner

With so many potential combinations of hyperparameters to tune, coming up with the best model can be a tedious process. Often two or more parameters might have correlated effects on the overall speed of convergence as well as validation accuracy, so tuning one at a time might not lead to the best model. And if curiosity gets the best of us, we might want to experiment on all the hyperparameters together.

Keras Tuner comes in to automate this hyperparameter search. We define a search algorithm, the potential values that each parameter can take (e.g., discrete values or a range), our target object to maximize (e.g., validation accuracy), and sit back to see the program start training. Keras Tuner conducts multiple experiments changing the parameters on our behalf, storing the metadata of the best model. The following code example adapted from Keras Tuner documentation showcases searching through the different model architectures (varying in the number of layers between 2 and 10) as well as varying the learning rate (between 0.1 and 0.001):

```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np

from kerastuner.engine.hypermodel import HyperModel
from kerastuner.engine.hyperparameters import HyperParameters

# Input data
(x, y), (val_x, val_y) = keras.datasets.mnist.load_data()
x = x.astype('float32') / 255.
val_x = val_x.astype('float32') / 255.

# Defining hyper parameters
hp = HyperParameters()
hp.Choice('learning_rate', [0.1, 0.001])
hp.Int('num_layers', 2, 10)

# Defining model with expandable number of layers
def build_model(hp):
    model = keras.Sequential()
    model.add(layers.Flatten(input_shape=(28, 28)))
    for _ in range(hp.get('num_layers')):
        model.add(layers.Dense(32, activation='relu'))
```

```
    model.add(layers.Dense(10, activation='softmax'))
    model.compile(
        optimizer=keras.optimizers.Adam(hp.get('learning_rate')),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
    return model

hypermodel = RandomSearch(
             build_model,
             max_trials=20, # Number of combinations allowed
             hyperparameters=hp,
             allow_new_entries=False,
             objective='val_accuracy')

hypermodel.search(x=x,
             y=y,
             epochs=5,
             validation_data=(val_x, val_y))

# Show summary of overall best model
hypermodel.results_summary()
```

Each experiment will show values like this:

```
 > Hp values:
  |-learning_rate: 0.001
  |-num_layers: 6
┌──────────────┬────────────┬──────────────┐
│ Name         │ Best model │ Current model │
├──────────────┼────────────┼──────────────┤
│ accuracy     │ 0.9911     │ 0.9911       │
│ loss         │ 0.0292     │ 0.0292       │
│ val_loss     │ 0.227      │ 0.227        │
│ val_accuracy │ 0.9406     │ 0.9406       │
└──────────────┴────────────┴──────────────┘
```

On the experiment end, the result summary gives a snapshot of the experiments conducted so far, and saves more metadata.

```
Hypertuning complete - results in ./untitled_project
[Results summary]
 |-Results in ./untitled_project
 |-Ran 20 trials
 |-Ran 20 executions (1 per trial)
 |-Best val_accuracy: 0.9406
```

Another big benefit is the ability to track experiments online in real time and get notifications on their progress by visiting *http://keras-tuner.appspot.com*, getting an API key (from Google App Engine), and entering the following line in our Python program along with the real API key:

```
tuner.enable_cloud(api_key=api_key)
```

Due to the potentially large combinatorial space, random search is preferred to grid search as a more practical way to get to a good solution on a limited experimentation budget. But there are faster ways, including Hyperband (Lisha Li et al.), whose implementation is also available in Keras Tuner.

For computer-vision problems, Keras Tuner includes ready-to-use tunable applications like HyperResNet.

Another big benefit is the ability to track experiments online in real time and get notifications on their progress by visiting *http://keras-tuner.appspot.com*, getting an API key (from Google App Engine), and entering the following line in our Python program along with the real API key:

```
tuner.enable_cloud(api_key=api_key)
```

Due to the potentially large combinatorial space, random search is preferred to grid search as a more practical way to get to a good solution on a limited experimentation budget. But there are faster ways, including Hyperband (Lisha Li et al.), whose implementation is also available in Keras Tuner.

For computer-vision problems, Keras Tuner includes ready-to-use tunable applications like HyperResNet.

AutoAugment

Another example hyperparameter are augmentations. Which augmentations to use? How much magnitude to augment? Would combining one too many make matters worse? Instead of leaving these decisions to humans, we can let AI decide. AutoAugment utilizes reinforcement learning to come up with the best combination of augmentations (like translation, rotation, shearing) and the probabilities and magnitudes to apply, to maximize the validation accuracy. (The method was applied by Ekin D. Cubuk et al. to come up with the new state-of-the-art ImageNet validation numbers.) By learning the best combination of augmentation parameters on ImageNet, we can readily apply it to our problem.

Applying the prelearned augmentation strategy from ImageNet is pretty simple:

```
from PIL import Image
from autoaugment import ImageNetPolicy
img = Image.open("cat.jpg")
policy = ImageNetPolicy()
imgs = [policy(img) for _ in range(8) ]
```

Figure 5-18 displays the results.

Figure 5-18. Output of augmentation strategies learned by reinforcement learning on the ImageNet datase

t

AutoKeras

With AI automating more and more jobs, it is no surprise it can finally automate designing AI architectures, too. NAS approaches utilize reinforcement learning to join together mini-architectural blocks until they are able to maximize the objective function; in other words, our validation accuracy. The current state-of-the-art networks are all based on NAS, leaving human-designed architectures in the dust. Research in this area started showing promising results in 2017, with a bigger focus on making train faster in 2018. And now with AutoKeras (Haifeng Jin et al.), we can also apply this state-of-the-art technique on our particular datasets in a relatively accessible manner.

Generating new model architectures with AutoKeras is a matter of supplying our images and associated labels as well as a time limit by which to finish running the jobs. Internally, it implements several optimization algorithms, including a Bayesian optimization approach to search for an optimal architecture:

```
!pip3 install autokeras
!pip3 install graphviz
from keras.datasets import mnist
from autokeras.image.image_supervised import ImageClassifier

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.reshape(x_test.shape + (1,))

clf = ImageClassifier(path=".",verbose=True, augment=False)
clf.fit(x_train, y_train, time_limit= 30 * 60) # 30 minutes
clf.final_fit(x_train, y_train, x_test, y_test, retrain=True)
y = clf.evaluate(x_test, y_test)
print(y)

# Save the model as a pickle file
clf.export_autokeras_model("model.pkl")

visualize('.')
```

Post-training, we are all eager to learn how the new model architecture looks. Unlike most of the cleaner-looking images we generally get to see, this will look pretty obfuscated to understand or print out. But what we do find faith in is that it yields high accuracy.

Summary

In this chapter, we saw a range of tools and techniques to help investigate opportunities to improve our CNN accuracy. Building a case for iterative experimentation, you learned how tuning hyperparameters can bring about optimal performance. And with so many hyperparameters to choose from, we then looked at automated approaches, including AutoKeras, AutoAugment, and Keras Tuner. Best of all, the core code for this chapter combining multiple tools in a single Colab file is available online on the book's GitHub (see *http://PracticalDeepLearning.ai*) and can easily be tuned to more than 100 datasets with a single line change and run online in the browser. Additionally, we compiled a checklist of actionable tips along with interactive experiments hosted online to help give your model a little extra edge. We hope that the material covered in this chapter will help make your models more robust, reduce bias, make them more explainable, and ultimately contribute to the responsible development of AI.

Summary

In this chapter, we saw a range of tools and techniques to help investigate opportunities to improve our CNN accuracy. Building a case for iterative experimentation, you learned how tuning hyperparameters can bring about optimal performance. And with so many hyperparameters to choose from, we then looked at automated approaches, including AutoKeras, AutoAugment, and Keras Tuner. Best of all, the core code for this chapter combining multiple tools in a single Colab file is available online on the book's GitHub (see *http://PracticalDeepLearning.ai*) and can easily be tuned to more than 100 datasets with a single line change and run online in the browser. Additionally, we compiled a checklist of actionable tips along with interactive experiments hosted online to help give your model a little extra edge. We hope that the material covered in this chapter will help make your models more robust, reduce bias, make them more explainable, and ultimately contribute to the responsible development of AI.

Chapter 6. Maximizing Speed and Performance of TensorFlow: A Handy Checklist

Life is all about making do with what we have, and optimization is the name of the game.

It's not about having everything—it's about using your resources wisely. Maybe we really want to buy that Ferrari, but our budget allows for a Toyota. You know what, though? With the right kinds of performance tuning, we can make that bad boy race at NASCAR!

Let's look at this in terms of the deep learning world. Google, with its engineering might and TPU pods capable of boiling the ocean, set a speed record by training ImageNet in just about 30 minutes! And yet, just a few months later, a ragtag team of three researchers (Andrew Shaw, Yaroslav Bulatov, and Jeremy Howard), with $40 in their pockets using a public cloud, were able to train ImageNet in only 18 minutes!

The lesson we can draw from these examples is that the amount of resources that you have is not nearly as important as using them to their maximum potential. It's all about doing more with less. In that spirit, this chapter is meant to serve as a handy checklist of potential performance optimizations that we can make when building all stages of the deep learning pipelines, and will be useful throughout the book. Specifically, we will discuss optimizations related to data preparation, data reading, data augmentation, training, and finally inference.

And the story starts and ends with two words...

Chapter 6. Maximizing Speed and Performance of TensorFlow: A Handy Checklist

Life is all about making do with what we have, and optimization is the name of the game.

It's not about having everything—it's about using your resources wisely. Maybe we really want to buy that Ferrari, but our budget allows for a Toyota. You know what, though? With the right kinds of performance tuning, we can make that bad boy race at NASCAR!

Let's look at this in terms of the deep learning world. Google, with its engineering might and TPU pods capable of boiling the ocean, set a speed record by training ImageNet in just about 30 minutes! And yet, just a few months later, a ragtag team of three researchers (Andrew Shaw, Yaroslav Bulatov, and Jeremy Howard), with $40 in their pockets using a public cloud, were able to train ImageNet in only 18 minutes!

The lesson we can draw from these examples is that the amount of resources that you have is not nearly as important as using them to their maximum potential. It's all about doing more with less. In that spirit, this chapter is meant to serve as a handy checklist of potential performance optimizations that we can make when building all stages of the deep learning pipelines, and will be useful throughout the book. Specifically, we will discuss optimizations related to data preparation, data reading, data augmentation, training, and finally inference.

And the story starts and ends with two words...

GPU Starvation

A commonly asked question by AI practitioners is, “Why is my training so slow?” The answer more often than not is GPU starvation.

GPUs are the lifelines of deep learning. They can also be the most expensive component in a computer system. In light of that, we want to fully utilize them. This means that a GPU should not need to wait for data to be available from other components for processing. Rather, when the GPU is ready to process, the preprocessed data should already be available at its doorstep and ready to go. Yet, the reality is that the CPU, memory, and storage are frequently the performance bottlenecks, resulting in suboptimal utilization of the GPU. In other words, we want the GPU to be the bottleneck, not the other way round.

Buying expensive GPUs for thousands of dollars can be worthwhile, but only if the GPU is the bottleneck to begin with. Otherwise, we might as well burn the cash.

To illustrate this better, consider Figure 6-1. In a deep learning pipeline, the CPU and GPU work in collaboration, passing data to each other. The CPU reads the data, performs preprocessing steps including augmentations, and then passes it on to the GPU for training. Their collaboration is like a relay race, except one of the relay runners is an Olympic athlete, waiting for a high school track runner to pass the baton. The more time the GPU stays idle, the more wasted resources.

Figure 6-1. GPU starvation, while waiting for CPU to finish preparing the data

A large portion of this chapter is devoted to reducing the idle time of the GPU and the CPU.

A logical question to ask is: how do we know whether the GPU is starving? Two handy tools can help us answer this question:

`nvidia-smi`

This command shows GPU statistics including utilization.

TensorFlow Profiler + TensorBoard

This visualizes program execution interactively in a timeline within TensorBoard.

TensorFlow Profiler + TensorBoard

This visualizes program execution interactively in a timeline within TensorBoard.

nvidia-smi

Short for NVIDIA System Management Interface program, `nvidia-smi` provides detailed statistics about our precious GPUs, including memory, utilization, temperature, power wattage, and more. It's a geek's dream come true

Let's take it for a test drive:

```
$ nvidia-smi
```

Figure 6-2 shows the result.

Figure 6-2. Terminal output of `nvidia-smi` *highlighting the GPU utilization*

While training a network, the key figure we are interested in is the GPU utilization, defined in the documentation as the percent of time over the past second during which *one or more* kernels was executing on the GPU. Fifty-one percent is frankly not that great. But this is utilization at the moment in time when `nvidia-smi` is called. How do we continuously monitor these numbers? To better understand the GPU usage, we can refresh the utilization metrics every half a second with the `watch` command (it's worth memorizing this command):

```
$ watch -n .5 nvidia-smi
```

NOTE

Although GPU utilization is a good proxy for measuring the efficiency of our pipeline, it does not alone measure how well we're using the GPU, because the work could still be using a small fraction of the GPU's resources.

Because staring at the terminal screen with the number jumping around is not the most optimal way to analyze, we can instead poll the GPU utilization every second and dump that into a file. Run this for about 30 seconds while any GPU-related process is running on our system and stop it by pressing Ctrl+C:

```
$ nvidia-smi --query-gpu=utilization.gpu --format=csv,noheader,nounits -f
gpu_utilization.csv -l 1
```

Now, calculate the median GPU utilization from the file generated:

```
$ sort -n gpu_utilization.csv | grep -v '^0$' | datamash median 1
```

TIP

Datamash is a handy command-line tool that performs basic numeric, textual, and statistical operations on textual data files. You can find instructions to install it at *https://www.gnu.org/software/datamash/*.

`nvidia-smi` is the most convenient way to check our GPU utilization on the command line. Could we get a deeper analysis? It turns out, for advanced users, TensorFlow provides a powerful set of tools.

NOTE

Although GPU utilization is a good proxy for measuring the efficiency of our pipeline, it does not alone measure how well we're using the GPU, because the work could still be using a small fraction of the GPU's resources.

Because staring at the terminal screen with the number jumping around is not the most optimal way to analyze, we can instead poll the GPU utilization every second and dump that into a file. Run this for about 30 seconds while any GPU-related process is running on our system and stop it by pressing Ctrl+C:

```
$ nvidia-smi --query-gpu=utilization.gpu --format=csv,noheader,nounits -f
gpu_utilization.csv -l 1
```

Now, calculate the median GPU utilization from the file generated:

```
$ sort -n gpu_utilization.csv | grep -v '^0$' | datamash median 1
```

TIP

Datamash is a handy command-line tool that performs basic numeric, textual, and statistical operations on textual data files. You can find instructions to install it at *https://www.gnu.org/software/datamash/*.

`nvidia-smi` is the most convenient way to check our GPU utilization on the command line. Could we get a deeper analysis? It turns out, for advanced users, TensorFlow provides a powerful set of tools.

TensorFlow Profiler + TensorBoard

TensorFlow ships with `tfprof` (Figure 6-3), the TensorFlow profiler to help analyze and understand the training process at a much deeper level, such as generating a detailed model analysis report for each operation in our model. But the command line can be a bit daunting to navigate. Luckily, TensorBoard, a suite of browser-based visualization tools for TensorFlow, includes a plugin for the profiler that lets us interactively debug the network with a few mouse clicks. This includes Trace Viewer, a feature that shows events in a timeline. It helps investigate how resources are being used precisely at a given period of time and spot inefficiencies.

> **NOTE**
>
> As of this writing, TensorBoard is fully supported only in Google Chrome and might not show the profile view in other browsers, like Firefox.

Figure 6-3. Profiler's timeline in TensorBoard shows an idle GPU while the CPU is processing as well as CPU idling while the GPU is processing

TensorBoard, by default, has the profiler enabled. Activating TensorBoard involves a simple callback function:

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="/tmp",
                                                       profile_batch=7)

model.fit(train_data,
          steps_per_epoch=10,
          epochs=2,
          callbacks=[tensorboard_callback])
```

When initializing the callback, unless `profile_batch` is explicitly specified, it

profiles the second batch. Why the second batch? Because the first batch is usually slower than the rest due to some initialization overhead.

NOTE

It bears reiterating that profiling using TensorBoard is best suited for power users of TensorFlow. If you are just starting out, you are better off using `nvidia-smi`. (Although `nvidia-smi` is a far more capable than just providing GPU utilization info, which is typically how most practitioners use it.) For users wanting even deeper access to their hardware utilization metrics, NVIDIA Nsight is a great tool.

Alright. With these tools at our disposal, we know that our program needs some tuning and has room for efficiency improvements. We look at those areas one by one in the next few sections.

profiles the second batch. Why the second batch? Because the first batch is usually slower than the rest due to some initialization overhead.

> **NOTE**
>
> It bears reiterating that profiling using TensorBoard is best suited for power users of TensorFlow. If you are just starting out, you are better off using `nvidia-smi`. (Although `nvidia-smi` is a far more capable than just providing GPU utilization info, which is typically how most practitioners use it.) For users wanting even deeper access to their hardware utilization metrics, NVIDIA Nsight is a great tool.

Alright. With these tools at our disposal, we know that our program needs some tuning and has room for efficiency improvements. We look at those areas one by one in the next few sections.

How to Use This Checklist

In business, an oft-quoted piece of advice is “You can’t improve what you can’t measure.” This applies to deep learning pipelines, as well. Tuning performance is like a science experiment. You set up a baseline run, tune a knob, measure the effect, and iterate in the direction of improvement. The items on the following checklist are our knobs—some are quick and easy, whereas others are more involved.

To use this checklist effectively, do the following:

1. Isolate the part of the pipeline that you want to improve.
2. Find a relevant point on the checklist.
3. Implement, experiment, and observe if runtime is reduced. If not reduced, ignore change.
4. Repeat steps 1 through 3 until the checklist is exhausted.

Some of the improvements might be minute, some more drastic. But the cumulative effect of all these changes should hopefully result in faster, more efficient execution and best of all, more bang for the buck for your hardware. Let’s look at each area of the deep learning pipeline step by step, including data preparation, data reading, data augmentation, training, and, finally, inference.

How to Use This Checklist

In business, an oft-quoted piece of advice is “You can’t improve what you can’t measure.” This applies to deep learning pipelines, as well. Tuning performance is like a science experiment. You set up a baseline run, tune a knob, measure the effect, and iterate in the direction of improvement. The items on the following checklist are our knobs—some are quick and easy, whereas others are more involved.

To use this checklist effectively, do the following:

1. Isolate the part of the pipeline that you want to improve.
2. Find a relevant point on the checklist.
3. Implement, experiment, and observe if runtime is reduced. If not reduced, ignore change.
4. Repeat steps 1 through 3 until the checklist is exhausted.

Some of the improvements might be minute, some more drastic. But the cumulative effect of all these changes should hopefully result in faster, more efficient execution and best of all, more bang for the buck for your hardware. Let’s look at each area of the deep learning pipeline step by step, including data preparation, data reading, data augmentation, training, and, finally, inference.

Performance Checklist

Performance Checklist

Data Preparation

- “Store as TFRecords”
- “Reduce Size of Input Data”
- “Use TensorFlow Datasets”

Data Preparation

- “Store as TFRecords”
- “Reduce Size of Input Data”
- “Use TensorFlow Datasets”

Data Reading

- “Use tf.data”
- “Prefetch Data”
- “Parallelize CPU Processing”
- “Parallelize I/O and Processing”
- “Enable Nondeterministic Ordering”
- “Cache Data”
- “Turn on Experimental Optimizations”
- “Autotune Parameter Values”

Data Reading

- “Use tf.data”
- “Prefetch Data”
- “Parallelize CPU Processing”
- “Parallelize I/O and Processing”
- “Enable Nondeterministic Ordering”
- “Cache Data”
- “Turn on Experimental Optimizations”
- “Autotune Parameter Values”

Data Augmentation

- “Use GPU for Augmentation”

Data Augmentation

- “Use GPU for Augmentation”

Training

- “Use Automatic Mixed Precision”
- “Use Larger Batch Size”
- “Use Multiples of Eight”
- “Find the Optimal Learning Rate”
- “Use tf.function”
- “Overtrain, and Then Generalize”
    - “Use progressive sampling”
    - “Use progressive augmentation”
    - “Use progressive resizing”
- “Install an Optimized Stack for the Hardware”
- “Optimize the Number of Parallel CPU Threads”
- “Use Better Hardware”
- “Distribute Training”
- “Examine Industry Benchmarks”

Training

- “Use Automatic Mixed Precision”
- “Use Larger Batch Size”
- “Use Multiples of Eight”
- “Find the Optimal Learning Rate”
- “Use tf.function”
- “Overtrain, and Then Generalize”
    - “Use progressive sampling”
    - “Use progressive augmentation”
    - “Use progressive resizing”
- “Install an Optimized Stack for the Hardware”
- “Optimize the Number of Parallel CPU Threads”
- “Use Better Hardware”
- “Distribute Training”
- “Examine Industry Benchmarks”

Inference

- “Use an Efficient Model”
- “Quantize the Model”
- “Prune the Model”
- “Use Fused Operations”
- “Enable GPU Persistence”

NOTE

A printable version of this checklist is available at http://PracticalDeepLearning.ai. Feel free to use it as a reference next time you train or deploy a model. Or even better, spread the cheer by sharing with your friends, colleagues, and more importantly, your manager.

Inference

- “Use an Efficient Model”
- “Quantize the Model”
- “Prune the Model”
- “Use Fused Operations”
- “Enable GPU Persistence”

NOTE

A printable version of this checklist is available at http://PracticalDeepLearning.ai. Feel free to use it as a reference next time you train or deploy a model. Or even better, spread the cheer by sharing with your friends, colleagues, and more importantly, your manager.

Data Preparation

There are a few optimizations that we can make even before we do any kind of training, and they have to do with how we prepare our data.

Data Preparation

There are a few optimizations that we can make even before we do any kind of training, and they have to do with how we prepare our data.

Store as TFRecords

Image datasets typically consist of thousands of tiny files, each file measuring a few kilobytes. And our training pipeline must read each file individually. Doing this thousands of times has significant overhead, causing a slowdown of the training process. That problem is even more severe in the case of spinning hard drives, for which the magnetic head needs to seek to the beginning of each file. This problem is further exacerbated when the files are stored on a remote storage service like the cloud. And there lies our first hurdle!

To speed up the reads, one idea is to combine thousands of files into a handful of larger files. And that's exactly what TFRecord does. It stores data in efficient Protocol Buffer (protobuf) objects, making them quicker to read. Let's see how to create TFRecord files:

```
# Create TFRecord files

import tensorflow as tf
from PIL import Image
import numpy as np
import io

cat = "cat.jpg"
img_name_to_labels = {'cat' : 0}
img_in_string = open(cat, 'rb').read()
label_for_img = img_name_to_labels['cat']

def getTFRecord(img, label):
 feature = {
    'label': _int64_feature(label),
    'image_raw': _bytes_feature(img),
 }
 return tf.train.Example(features=tf.train.Features(feature=feature))

with tf.compat.v1.python_io.TFRecordWriter('img.tfrecord') as writer:
  for filename, label in img_name_to_labels.items():
    image_string = open(filename, 'rb').read()
    tf_example = getTFRecord(image_string, label)
    writer.write(tf_example.SerializeToString())
```

Now, let's take a look at reading these TFRecord files:

```python
# Reading TFRecord files

dataset = tf.data.TFRecordDataset('img.tfrecord')
ground_truth_info = {
  'label': tf.compat.v1.FixedLenFeature([], tf.int64),
  'image_raw': tf.compat.v1.FixedLenFeature([], tf.string),
}

def map_operation(read_data):
  return tf.compat.v1.parse_single_example(read_data, ground_truth_info)

imgs = dataset.map(map_operation)

for image_features in imgs:
  image_raw = image_features['image_raw'].numpy()
  label = image_features['label'].numpy()
  image = Image.open(io.BytesIO(image_raw))
  image.show()
  print(label)
```

So, why not join all of the data in a single file, like say for ImageNet? Although reading thousands of tiny files harms performance due to the overhead involved, reading gigantic files is an equally bad idea. They reduce our ability to make parallel reads and parallel network calls. The sweet spot to shard (divide) a large dataset in TFRecord files lies at around 100 MB.

```python
# Reading TFRecord files

dataset = tf.data.TFRecordDataset('img.tfrecord')
ground_truth_info = {
  'label': tf.compat.v1.FixedLenFeature([], tf.int64),
  'image_raw': tf.compat.v1.FixedLenFeature([], tf.string),
}

def map_operation(read_data):
  return tf.compat.v1.parse_single_example(read_data, ground_truth_info)

imgs = dataset.map(map_operation)

for image_features in imgs:
  image_raw = image_features['image_raw'].numpy()
  label = image_features['label'].numpy()
  image = Image.open(io.BytesIO(image_raw))
  image.show()
  print(label)
```

So, why not join all of the data in a single file, like say for ImageNet? Although reading thousands of tiny files harms performance due to the overhead involved, reading gigantic files is an equally bad idea. They reduce our ability to make parallel reads and parallel network calls. The sweet spot to shard (divide) a large dataset in TFRecord files lies at around 100 MB.

Reduce Size of Input Data

Image datasets with large images need to be resized before passing through to the GPU. This means the following:

- Repeated CPU cycles at every iteration
- Repeated I/O bandwidth being consumed at a larger rate than needed in our data pipeline

One good strategy to save compute cycles is to perform common preprocessing steps once on the entire dataset (like resizing) and then saving the results in TFRecord files for all future runs.

Reduce Size of Input Data

Image datasets with large images need to be resized before passing through to the GPU. This means the following:

- Repeated CPU cycles at every iteration
- Repeated I/O bandwidth being consumed at a larger rate than needed in our data pipeline

One good strategy to save compute cycles is to perform common preprocessing steps once on the entire dataset (like resizing) and then saving the results in TFRecord files for all future runs.

Use TensorFlow Datasets

For commonly used public datasets, from MNIST (11 MB) to CIFAR-100 (160 MB) all the way to MS COCO (38 GB) and Google Open Images (565 GB), it's quite an effort to download the data (often spread across multiple zipped files). Imagine your frustration if after downloading 95% of the file slowly, the connection becomes spotty and breaks. This is not unusual because these files are typically hosted on university servers, or are downloaded from various sources like Flickr (as is the case with ImageNet 2012, which gives us the URLs from which to download 150 GB-plus of images). A broken connection might mean having to start all over again.

If you think that was tedious, the real challenge actually begins only after you successfully download the data. For every new dataset, we now need to hunt through the documentation to determine how the data is formatted and organized, so we can begin reading and processing appropriately. Then, we need to split the data into training, validation, and test sets (preferably converting to TFRecords). And when the data is so large as to not fit in memory, we will need to do some manual jiu-jitsu to read it and feed it efficiently to the training pipeline. We never said it was easy.

Alternately, we could skip all the pain by consuming the high-performance, ready-to-use TensorFlow Datasets package. With several famous datasets available, it downloads, splits, and feeds our training pipeline using best practices in a few lines.

Let's look at which datasets are available.

```
import tensorflow_datasets as tfds

# See available datasets
print(tfds.list_builders())
```

```
===== Output =====
['abstract_reasoning', 'bair_robot_pushing_small', 'caltech101', 'cats_vs_dogs',
'celeb_a', 'celeb_a_hq', 'chexpert', 'cifar10', 'cifar100', 'cifar10_corrupted',
'cnn_dailymail', 'coco2014', 'colorectal_histology',
'colorectal_histology_large', 'cycle_gan' ...
```

There are more than 100 datasets as of this writing, and that number is steadily increasing. Now, let's download, extract, and make an efficient pipeline using the training set of CIFAR-10:

```
train_dataset = tfds.load(name="cifar100", split=tfds.Split.TRAIN)
train_dataset = train_dataset.shuffle(2048).batch(64)
```

That's it! The first time we execute the code, it will download and cache the dataset on our machine. For every future run, it will skip the network download and directly read from the cache.

There are more than 100 datasets as of this writing, and that number is steadily increasing. Now, let's download, extract, and make an efficient pipeline using the training set of CIFAR-10:

```
train_dataset = tfds.load(name="cifar100", split=tfds.Split.TRAIN)
train_dataset = train_dataset.shuffle(2048).batch(64)
```

That's it! The first time we execute the code, it will download and cache the dataset on our machine. For every future run, it will skip the network download and directly read from the cache.

Data Reading

Now that the data is prepared, let’s look for opportunities to maximize the throughput of the data reading pipeline.

Data Reading

Now that the data is prepared, let’s look for opportunities to maximize the throughput of the data reading pipeline.

Use tf.data

We could choose to manually read every file from our dataset with Python's built-in I/O library. We could simply call `open` for each file and we'd be good to go, right? The main downside in this approach is that our GPU would be bottlenecked by our file reads. Every time we read a file, the GPU needs to wait. Every time the GPU starts processing its input, we wait before we read the next file from disk. Seems rather wasteful, doesn't it?

If there's only one thing you can take away from this chapter, let it be this: `tf.data` is the way to go for building a high-performance training pipeline. In the next few sections, we explore several aspects of `tf.data` that you can exploit to improve training speed.

Let's set up a base pipeline for reading data:

```
files = tf.data.Dataset.list_files("./training_data/*.tfrecord")
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(2048)
                 .repeat()
                 .map(lambda item: tf.io.parse_single_example(item, features))
                 .map(_resize_image)
                 .batch(64)
```

Use tf.data

We could choose to manually read every file from our dataset with Python's built-in I/O library. We could simply call `open` for each file and we'd be good to go, right? The main downside in this approach is that our GPU would be bottlenecked by our file reads. Every time we read a file, the GPU needs to wait. Every time the GPU starts processing its input, we wait before we read the next file from disk. Seems rather wasteful, doesn't it?

If there's only one thing you can take away from this chapter, let it be this: `tf.data` is the way to go for building a high-performance training pipeline. In the next few sections, we explore several aspects of `tf.data` that you can exploit to improve training speed.

Let's set up a base pipeline for reading data:

```
files = tf.data.Dataset.list_files("./training_data/*.tfrecord")
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(2048)
                 .repeat()
                 .map(lambda item: tf.io.parse_single_example(item, features))
                 .map(_resize_image)
                 .batch(64)
```

Prefetch Data

In the pipeline we discussed earlier, the GPU waits for the CPU to generate data, and then the CPU waits for the GPU to finish computation before generating data for the next cycle. This circular dependency causes idle time for both CPU and GPU, which is inefficient.

The `prefetch` function helps us here by delinking the production of the data (by the CPU) from the consumption of the data (by the GPU). Using a background thread, it allows data to be passed *asynchronously* into an intermediate buffer, where it is readily available for a GPU to consume. The CPU now carries on with the next computation instead of waiting for the GPU. Similarly, as soon as the GPU is finished with its previous computation, and there's data readily available in the buffer, it starts processing.

To use it, we can simply call `prefetch` on our dataset at the very end of our pipeline along with a `buffer_size` parameter (which is the maximum amount o data that can be stored). Usually `buffer_size` is a small number; `1` is good enough in many cases:

```
dataset = dataset.prefetch(buffer_size=16)
```

In just a few pages, we show you how to find an optimal value for this parameter.

In summary, if there's an opportunity to overlap CPU and GPU computations, `prefetch` will automatically exploit it.

Prefetch Data

In the pipeline we discussed earlier, the GPU waits for the CPU to generate data, and then the CPU waits for the GPU to finish computation before generating data for the next cycle. This circular dependency causes idle time for both CPU and GPU, which is inefficient.

The `prefetch` function helps us here by delinking the production of the data (by the CPU) from the consumption of the data (by the GPU). Using a background thread, it allows data to be passed *asynchronously* into an intermediate buffer, where it is readily available for a GPU to consume. The CPU now carries on with the next computation instead of waiting for the GPU. Similarly, as soon as the GPU is finished with its previous computation, and there's data readily available in the buffer, it starts processing.

To use it, we can simply call `prefetch` on our dataset at the very end of our pipeline along with a `buffer_size` parameter (which is the maximum amount of data that can be stored). Usually `buffer_size` is a small number; `1` is good enough in many cases:

```
dataset = dataset.prefetch(buffer_size=16)
```

In just a few pages, we show you how to find an optimal value for this parameter.

In summary, if there's an opportunity to overlap CPU and GPU computations, `prefetch` will automatically exploit it.

Parallelize CPU Processing

It would be a waste to have a CPU with multiple cores but doing all of our processing on only one of them. Why not take advantage of the rest? This is exactly where the `num_parallel_calls` argument in the `map` function comes in handy:

```
dataset = dataset.map(lambda item: tf.io.parse_single_example(item, features),
                      num_parallel_calls=4)
```

This starts multiple threads to parallelize processing of the `map()` function. Assuming that there is no heavy application running in the background, we will want to set `num_parallel_calls` to the number of CPU cores on our system. Anything more will potentially degrade the performance due to the overhead of context switching.

Parallelize CPU Processing

It would be a waste to have a CPU with multiple cores but doing all of our processing on only one of them. Why not take advantage of the rest? This is exactly where the `num_parallel_calls` argument in the `map` function comes in handy:

```
dataset = dataset.map(lambda item: tf.io.parse_single_example(item, features),
                      num_parallel_calls=4)
```

This starts multiple threads to parallelize processing of the `map()` function. Assuming that there is no heavy application running in the background, we will want to set `num_parallel_calls` to the number of CPU cores on our system. Anything more will potentially degrade the performance due to the overhead of context switching.

Parallelize I/O and Processing

Reading files from disk or worse, over a network, is a huge cause of bottlenecks. We might possess the best CPU and GPU in the world, but if we don't optimize our file reads, it would all be for naught. One solution that addresses this problem is to parallelize both I/O and subsequent processing (also known as *interleaving*).

```
dataset = files.interleave(map_func, num_parallel_calls=4)
```

In this command, two things are happening:

- The input data is acquired in parallel (by default equal to the number of cores on the system).
- On the acquired data, setting the `num_parallel_calls` parameter allows the `map_func` function to execute on multiple parallel threads and read from the incoming data asynchronously.

If `num_parallel_calls` was not specified, even if the data were read in parallel `map_func` would run synchronously on a single thread. As long as `map_func` runs faster than the rate at which the input data is coming in, there will not be a problem. We definitely want to set `num_parallel_calls` higher if `map_func` becomes a bottleneck.

Parallelize I/O and Processing

Reading files from disk or worse, over a network, is a huge cause of bottlenecks. We might possess the best CPU and GPU in the world, but if we don't optimize our file reads, it would all be for naught. One solution that addresses this problem is to parallelize both I/O and subsequent processing (also known as *interleaving*).

```
dataset = files.interleave(map_func, num_parallel_calls=4)
```

In this command, two things are happening:

- The input data is acquired in parallel (by default equal to the number of cores on the system).
- On the acquired data, setting the `num_parallel_calls` parameter allows the `map_func` function to execute on multiple parallel threads and read from the incoming data asynchronously.

If `num_parallel_calls` was not specified, even if the data were read in parallel, `map_func` would run synchronously on a single thread. As long as `map_func` runs faster than the rate at which the input data is coming in, there will not be a problem. We definitely want to set `num_parallel_calls` higher if `map_func` becomes a bottleneck.

Enable Nondeterministic Ordering

For many datasets, the reading order is not important. After all, we might be randomizing their ordering anyway. By default, when reading files in parallel, `tf.data` still attempts to produce their outputs in a *fixed round-robin order*. The disadvantage is that we might encounter a "straggler" along the way (i.e., an operation that takes a lot longer than others, such as a slow file read, and holds up all other operations). It's like a grocery store line where the person in front of us insists on using cash with the exact change, whereas everyone else uses a credit card. So instead of blocking all the subsequent operations that are ready to give output, we skip over the stragglers until they are done with their processing This breaks the ordering while reducing wasted cycles waiting for the handful of slower operations:

```
options = tf.data.Options()
options.experimental_deterministic = False

dataset = tf.data.Dataset.list_files("./training_data/")
dataset = dataset.with_options(options)
dataset = dataset.interleave(tf.data.TFRecordDataset, num_parallel_calls=4)
```

Enable Nondeterministic Ordering

For many datasets, the reading order is not important. After all, we might be randomizing their ordering anyway. By default, when reading files in parallel, `tf.data` still attempts to produce their outputs in a *fixed round-robin order*. The disadvantage is that we might encounter a "straggler" along the way (i.e., an operation that takes a lot longer than others, such as a slow file read, and holds up all other operations). It's like a grocery store line where the person in front of us insists on using cash with the exact change, whereas everyone else uses a credit card. So instead of blocking all the subsequent operations that are ready to give output, we skip over the stragglers until they are done with their processing. This breaks the ordering while reducing wasted cycles waiting for the handful of slower operations:

```
options = tf.data.Options()
options.experimental_deterministic = False

dataset = tf.data.Dataset.list_files("./training_data/")
dataset = dataset.with_options(options)
dataset = dataset.interleave(tf.data.TFRecordDataset, num_parallel_calls=4)
```

Cache Data

The `Dataset.cache()` function allows us to make a copy of data either in memory or as a file on disk. There are two reasons why you might want to cache a dataset:

- To avoid repeatedly reading from disk after the first epoch. This is obviously effective only when the cache is in memory and can fit in the available RAM.
- To avoid having to repeatedly perform expensive CPU operations on data (e.g., resizing large images to a smaller size).

TIP

Cache is best used for data that is not going to change. It is recommended to place `cache()` before any random augmentations and shuffling; otherwise, caching at the end will result in exactly the same data and order in every run.

Depending on our scenario, we can use one of the two following lines:

```
dataset = dataset.cache()                    # in-memory
dataset = dataset.cache(filename='tmp.cache') # on-disk
```

It's worth noting that in-memory cache is volatile and hence only shows performance improvements in the second epoch of every run. On the other hand, file-based cache will make every run faster (beyond the very first epoch of the first run).

TIP

In the "Reduce Size of Input Data", we mentioned preprocessing the data and saving it as TFRecord files as input to future data pipelines. Using the **`cache()`** function directly after the preprocessing step in your pipeline would give a similar performance with a single word change in code.

Turn on Experimental Optimizations

TensorFlow has many built-in optimizations, often initially experimental and turned off by default. Depending on your use case, you might want to turn on some of them to squeeze out just a little more performance from your pipeline. Many of these optimizations are detailed in the documentation for `tf.data.experimental.OptimizationOptions`.

NOTE

Here's a quick refresher on filter and map operations:

Filter

A filter operation goes through a list element by element and grabs those that match a given condition. The condition is supplied as a lambda operation that returns a boolean value.

Map

A map operation simply takes in an element, performs a computation, and returns an output. For example, resizing an image.

Let's look at a few experimental optimizations that are available to us, including examples of two consecutive operations that could benefit from being fused together as one single operation.

Filter fusion

Sometimes, we might want to filter based on multiple attributes. Maybe we want to use only images that have both a dog and a cat. Or, in a census dataset, only look at families above a certain income threshold who also live within a certain distance to the city center. `filter_fusion` can help speed up such scenarios. Consider the following example:

```
dataset = dataset.filter(lambda x: x < 1000).filter(lambda x: x % 3 == 0)
```

The first filter performs a full pass over the entire dataset and returns elements that are less than 1,000. On this output, the second filter does another pass to further remove elements not divisible by three. Instead of doing two passes over many of the same elements, we could instead combine both the filter operations into one pass using an `AND` operation. That is precisely what the `filter_fusion` option enables—combining multiple filter operations into one pass. By default, i is turned off. You can enable it by using the following statement:

```
options = tf.data.Options()
options.experimental_optimization.filter_fusion = True
dataset = dataset.with_options(options)
```

Map and filter fusion

Consider the following example:

```
dataset = dataset.map(lambda x: x * x).filter(lambda x: x % 2 == 0)
```

In this example, the `map` function does a full pass on the entire dataset to calculate the square of every element. Then, the `filter` function discards the odd elements. Rather than doing two passes (more so in this particularly wasteful example), we could simply fuse the map and filter operations together by turning on the `map_and_filter_fusion` option so that they operate as a single unit:

```
options.experimental_optimization.map_and_filter_fusion = True
```

Map fusion

Similar to the aforementioned two examples, fusing two or more map operations prevents multiple passes from being performed on the same data and instead combines them in a single pass:

```
options.experimental_optimization.map_fusion = True
```

t

;

Autotune Parameter Values

You might have noticed that many of the code examples in this section have hardcoded values for some of the parameters. For the combination of the problem and hardware at hand, you can tune them for maximum efficiency. How to tune them? One obvious way is to manually tweak the parameters one at a time and isolate and observe the impact of each of them on the overall performance until we get the precise parameter set. But the number of knobs to tune quickly gets out of hand due to the combinatorial explosion. If this wasn't enough, our finely tuned script wouldn't necessarily be as efficient on another machine due to differences in hardware such as the number of CPU cores, GPU availability, and so on. And even on the same system, depending on resource usage by other programs, these knobs might need to be adjusted over different runs.

How do we solve this? We do the opposite of manual tuning: autotuning. Using hill-climbing optimization algorithms (which are a type of heuristic-driven search algorithms), this option automatically finds the ideal parameter combination for many of the `tf.data` function parameters. Simply use `tf.data.experimental.AUTOTUNE` instead of manually assigning numbers. It's the one parameter to rule them all. Consider the following example:

```
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

Isn't that an elegant solution? We can do that for several other function calls in the `tf.data` pipeline. The following is an example of combining together several optimizations from the section "Data Reading" to make a high-performance data pipeline:

```
options = tf.data.Options()
options.experimental_deterministic = False

dataset = tf.data.Dataset.list_files("/path/*.tfrecord")
dataset = dataset.with_options(options)
dataset = files.interleave(tf.data.TFRecordDataset,
                           num_parallel_calls=tf.data.experimental.AUTOTUNE)
dataset = dataset.map(preprocess,
                      num_parallel_calls=tf.data.experimental.AUTOTUNE)
```

```
    dataset = dataset.cache()
    dataset = dataset.repeat()
    dataset = dataset.shuffle(2048)
    dataset = dataset.batch(batch_size=64)
    dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

7

```
dataset = dataset.cache()
dataset = dataset.repeat()
dataset = dataset.shuffle(2048)
dataset = dataset.batch(batch_size=64)
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

Data Augmentation

Sometimes, we might not have sufficient data to run our training pipeline. Even if we did, we might still want to manipulate the images to improve the robustness of our model—with the help of data augmentation. Let’s see whether we can make this step any faster.

Data Augmentation

Sometimes, we might not have sufficient data to run our training pipeline. Even if we did, we might still want to manipulate the images to improve the robustness of our model—with the help of data augmentation. Let’s see whether we can make this step any faster.

Use GPU for Augmentation

Data preprocessing pipelines can be elaborate enough that you could write an entire book about them. Image transformation operations such as resizing, cropping, color transformations, blurring, and so on are commonly performed on the data immediately after it's read from disk into memory. Given that these are all matrix transformation operations, they might do well on a GPU.

OpenCV, Pillow, and the built-in Keras augmentation functionality are the most commonly used libraries in computer vision for working on images. There's one major limitation here, though. Their image processing is primarily CPU based (although you can compile OpenCV to work with CUDA), which means that the pipeline might not be fully utilizing the underlying hardware to its true potential.

> **NOTE**
>
> As of August 2019, there are efforts underway to convert Keras image augmentation to be GPU accelerated, as well.

There are a few different GPU-bound options that we can explore.

tf.image built-in augmentations

`tf.image` provides some handy augmentation functions that we can seamlessly plug into a `tf.data` pipeline. Some of the methods include image flipping, color augmentations (hue, saturation, brightness, contrast), zooming, and rotation. Consider the following example, which changes the hue of an image:

```
updated_image = tf.image.adjust_hue(image, delta = 0.2)
```

The downside to relying on `tf.image` is that the functionality is much more limited compared to OpenCV, Pillow, and even Keras. For example, the built-in function for image rotation in `tf.image` only supports rotating images by 90 degrees counter-clockwise. If we need to be able to rotate by an arbitrary amount, such as 10 degrees, we'd need to manually build that functionality. Keras, on the other hand, provides that functionality out of the box.

As another alternative to the `tf.data` pipeline, the NVIDIA Data Loading Library (DALI) offers a fast data loading and preprocessing pipeline accelerated by GPU processing. As shown in Figure 6-4, DALI implements several common steps including resizing an image and augmenting an image in the GPU, immediately before the training. DALI works with multiple deep learning frameworks including TensorFlow, PyTorch, MXNet, and others, offering portability of the preprocessing pipelines.

NVIDIA DALI

Figure 6-4. The NVIDIA DALI pipeline

Additionally, even JPEG decoding (a relatively heavy task) can partially make use of the GPU, giving it an additional boost. This is done using nvJPEG, a GPU-accelerated library for JPEG decoding. For multi-GPU tasks, this scales near linearly as the number of GPUs increases.

NVIDIAs efforts culminated in a record-breaking MLPerf entry (which benchmarks machine learning hardware, software, and services), training a ResNet-50 model in 80 seconds.

As another alternative to the `tf.data` pipeline, the NVIDIA Data Loading Library (DALI) offers a fast data loading and preprocessing pipeline accelerated by GPU processing. As shown in Figure 6-4, DALI implements several common steps including resizing an image and augmenting an image in the GPU, immediately before the training. DALI works with multiple deep learning frameworks including TensorFlow, PyTorch, MXNet, and others, offering portability of the preprocessing pipelines.

NVIDIA DALI

Figure 6-4. The NVIDIA DALI pipeline

Additionally, even JPEG decoding (a relatively heavy task) can partially make use of the GPU, giving it an additional boost. This is done using nvJPEG, a GPU-accelerated library for JPEG decoding. For multi-GPU tasks, this scales near linearly as the number of GPUs increases.

NVIDIAs efforts culminated in a record-breaking MLPerf entry (which benchmarks machine learning hardware, software, and services), training a ResNet-50 model in 80 seconds.

Training

For those beginning their performance optimization journey, the quickest wins come from improving the data pipelines, which is relatively easy. For a training pipeline that is already being fed data fast, let’s investigate optimizations for our actual training step.

Training

For those beginning their performance optimization journey, the quickest wins come from improving the data pipelines, which is relatively easy. For a training pipeline that is already being fed data fast, let’s investigate optimizations for our actual training step.

Use Automatic Mixed Precision

"*One line to make your training two to three times faster!*"

Weights in deep learning models are typically stored in single-precision; that is, 32-bit floating point, or as it's more commonly referenced: FP32. Putting these models in memory-constrained devices such as mobile phones can be challenging to accommodate. A simple trick to make models smaller is to convert them from single-precision (FP32) to half-precision (FP16). Sure, the representative power of these weights goes down, but as we demonstrate later in this chapter ("Quantize the Model"), neural networks are resilient to small changes, much like they are resilient to noise in images. Hence, we get the benefits of a more efficient model without sacrificing much accuracy. In fact, we can even reduce the representation to 8-bit integers (INT8) without a significant loss in accuracy, as we will see in some upcoming chapters.

So, if we can use reduced-precision representation during inference, could we do the same during training, as well? Going from 32-bit to 16-bit representation would effectively mean double the memory bandwidth available, double the model size, or double the batch size can be accommodated. Unfortunately, it turns out that using FP16 naïvely *during training* can potentially lead to a significant loss in model accuracy and might not even converge to an optimal solution. This happens because of FP16's limited range for representing numbers. Due to a lack of adequate precision, any updates to the model during training, if sufficiently small, will cause an update to not even register. Imagine adding 0.00006 to a weight value of 1.1. With FP32, the weight would be correctly updated to 1.10006. With FP16, however, the weight would remain 1.1 Conversely, any activations from layers such as Rectified Linear Unit (ReLU) could be high enough for FP16 to overflow and hit infinity (`NaN` in Python).

The easy answer to these challenges is to use automatic mixed-precision training. In this method, we store the model in FP32 as a master copy and perform the forward/backward passes of training in FP16. After each training step is performed, the final update from that step is then scaled back up to FP32 before it is applied to the master copy. This helps avoid the pitfalls of FP16 arithmetic and results in a lower memory footprint, and faster training (experiments have shown increases in speed by two to three times), while

achieving similar accuracy levels as training solely in FP32. It is noteworthy that newer GPU architectures like the NVIDIA Volta and Turing especially optimize FP16 operations.

To enable mixed precision during training, we simply need to add the following line to the beginning of our Python script:

```
os.environ['TF_ENABLE_AUTO_MIXED_PRECISION'] = '1'
```

achieving similar accuracy levels as training solely in FP32. It is noteworthy that newer GPU architectures like the NVIDIA Volta and Turing especially optimize FP16 operations.

To enable mixed precision during training, we simply need to add the following line to the beginning of our Python script:

```
os.environ['TF_ENABLE_AUTO_MIXED_PRECISION'] = '1'
```

Use Larger Batch Size

Instead of using the entire dataset for training in one batch, we train with several minibatches of data. This is done for two reasons:

- Our full data (single batch) might not fit in the GPU RAM.
- We can achieve similar training accuracy by feeding many smaller batches, just as you would by feeding fewer larger batches.

Having smaller minibatches might not fully utilize the available GPU memory, so it's vital to experiment with this parameter, see its effect on the GPU utilization (using the `nvidia-smi` command), and choose the batch size that maximizes the utilization. Consumer GPUs like the NVIDIA 2080 Ti ship with 11 GB of GPU memory, which is plenty for efficient models like MobileNet family.

For example on hardware with the 2080 Ti graphics card, using 224 x 224 resolution images and MobileNetV2 model, the GPU can accommodate a batch size up to 864. Figure 6-5 shows the effect of varying batch sizes from 4 to 864, on both the GPU utilization (solid line) as well as the time per epoch (dashed line). As we can see in the figure, the higher the batch size, the higher the GPU utilization, leading to a shorter training time per epoch.

Even at our max batch size of 864 (before running out of memory allocation), the GPU utilization does not cross 85%. This means that the GPU was fast enough to handle the computations of our otherwise very efficient data pipeline. Replacing MobileNetV2 with a heavier ResNet-50 model immediately increased GPU to 95%.

Figure 6-5. Effect of varying batch size on time per epoch (seconds) as well as on percentage GPU utilization (Log scales have been used for both X- and Y-axes.)

TIP

Even though we showcased batch sizes up to a few hundreds, large industrial training loads distributed across multiple nodes often use much larger batch sizes with the help of a technique called Layer-wise Adaptive Rate Scaling (LARS). For example, Fujitsu Research trained a ResNet-50 network to 75% Top-1 accuracy on ImageNet in a mere 75 seconds. Their ammunition? 2048 Tesla V100 GPUs and a whopping batch size of 81,920!

Figure 6-5. Effect of varying batch size on time per epoch (seconds) as well as on percentage GPU utilization (Log scales have been used for both X- and Y-axes.)

TIP

Even though we showcased batch sizes up to a few hundreds, large industrial training loads distributed across multiple nodes often use much larger batch sizes with the help of a technique called Layer-wise Adaptive Rate Scaling (LARS). For example, Fujitsu Research trained a ResNet-50 network to 75% Top-1 accuracy on ImageNet in a mere 75 seconds. Their ammunition? 2048 Tesla V100 GPUs and a whopping batch size of 81,920!

Use Multiples of Eight

Most of the computations in deep learning are in the form of “matrix multiply and add.” Although it’s an expensive operation, specialized hardware has increasingly been built in the past few years to optimize for its performance. Examples include Google’s TPUs and NVIDIA’s Tensor Cores (which can be found in the Turing and Volta architectures). Turing GPUs provide both Tensor Cores (for FP16 and INT8 operations) as well as CUDA cores (for FP32 operations), with the Tensor Cores delivering significantly higher throughput. Due to their specialized nature, Tensor Cores require that certain parameters within the data supplied to them be divisible by eight. Here are just three such parameters:

- The number of channels in a convolutional filter
- The number of neurons in a fully connected layer and the inputs to this layer
- The size of minibatches

If these parameters are not divisible by eight, the GPU CUDA cores will be used as the fallback accelerator instead. In an experiment reported by NVIDIA, simply changing the batch size from 4,095 to 4,096 resulted in an increase in throughput of five times. Keep in mind that using multiples of eight (or 16 in the case of INT8 operations), in addition to using automatic mixed precision, is the bare minimum requirement to activate the Tensor Cores. For higher efficiency, the recommended values are in fact multiples of 64 or 256. Similarly, Google recommends multiples of 128 when using TPUs for maximum efficiency.

Use Multiples of Eight

Most of the computations in deep learning are in the form of “matrix multiply and add.” Although it’s an expensive operation, specialized hardware has increasingly been built in the past few years to optimize for its performance. Examples include Google’s TPUs and NVIDIA’s Tensor Cores (which can be found in the Turing and Volta architectures). Turing GPUs provide both Tensor Cores (for FP16 and INT8 operations) as well as CUDA cores (for FP32 operations), with the Tensor Cores delivering significantly higher throughput. Due to their specialized nature, Tensor Cores require that certain parameters within the data supplied to them be divisible by eight. Here are just three such parameters:

- The number of channels in a convolutional filter
- The number of neurons in a fully connected layer and the inputs to this layer
- The size of minibatches

If these parameters are not divisible by eight, the GPU CUDA cores will be used as the fallback accelerator instead. In an experiment reported by NVIDIA, simply changing the batch size from 4,095 to 4,096 resulted in an increase in throughput of five times. Keep in mind that using multiples of eight (or 16 in the case of INT8 operations), in addition to using automatic mixed precision, is the bare minimum requirement to activate the Tensor Cores. For higher efficiency, the recommended values are in fact multiples of 64 or 256. Similarly, Google recommends multiples of 128 when using TPUs for maximum efficiency.

Find the Optimal Learning Rate

One hyperparameter that greatly affects our speed of convergence (and accuracy is the learning rate. The ideal result of training is the global minimum; that is, the point of least loss. Too high a learning rate can cause our model to overshoot the global minimum (like a wildly swinging pendulum) and potentially never converge. Too low a learning rate can cause convergence to take too long because the learning algorithm will take very small steps toward the minimum. Finding the right initial learning rate can make a world of difference.

The naive way to find the ideal initial learning rate is to try a few different learning rates (such as 0.00001, 0.0001, 0.001, 0.01, 0.1) and find one that starts converging quicker than others. Or, even better, perform grid search over a range of values. This approach has two problems: 1) depending on the granularity, it might find a decent value, but it might not be the most optimal value; and 2) we need to train multiple times, which can be time consuming.

In Leslie N. Smith's 2015 paper, "Cyclical Learning Rates for Training Neural Networks," he describes a much better strategy to find this optimal learning rate. In summary:

1. Start with a really low learning rate and gradually increase it until reaching a prespecified maximum value.
2. At each learning rate, observe the loss—first it will be stagnant, then it will begin going down and then eventually go back up.
3. Calculate the rate of decrease of loss (first derivative) at each learning rate.
4. Select the point with the highest rate of decrease of loss.

It sounds like a lot of steps, but thankfully we don't need to write code for it. The keras_lr_finder library by Pavel Surmenok gives us a handy function to find it:

```
lr_finder = LRFinder(model)
lr_finder.find(x_train, y_train, start_lr=0.0001, end_lr=10, batch_size=512,
               epochs=5)
lr_finder.plot_loss(n_skip_beginning=20, n_skip_end=5)
```

Figure 6-6 shows the plot of loss versus learning rate. It becomes evident that a learning rate of 10^{-4} or 10^{-3} might be too low (owing to barely any drop in loss), and similarly, above 1 might be too high (because of the rapid increase in loss).

Figure 6-6. A graph showing the change in loss as the learning rate is increased

What we are most interested in is the point of the greatest decrease in loss. After all, we want to minimize the time we spend in getting to the least loss during training. In Figure 6-7, we plot the *rate of change* of loss—the derivative of the loss with regard to the learning rate:

```
# Show Simple Moving Average over 20 points to smoothen the graph
lr_finder.plot_loss_change(sma=20, n_skip_beginning=20, n_skip_end=5,
                           y_lim=(-0.01, 0.01))
```

Figure 6-7. A graph showing the rate of change in loss as the learning rate is increased

These figures show that values around 0.1 would lead to the fastest decrease in loss, and hence we would choose it as our optimal learning rate.

These figures show that values around 0.1 would lead to the fastest decrease in loss, and hence we would choose it as our optimal learning rate.

Use tf.function

Eager execution mode, which is turned on by default in TensorFlow 2.0, allows users to execute code line by line and immediately see the results. This is immensely helpful in development and debugging. This is in contrast to TensorFlow 1.x, for which the user had to build all operations as a graph and then execute them in one go to see the results. This made debugging a nightmare!

Does the added flexibility from eager execution come at a cost? Yes, a tiny one, typically in the order of microseconds, which can essentially be ignored for large compute-intensive operations, like training ResNet-50. But where there are many small operations, eager execution can have a sizable impact.

We can overcome this by two approaches:

Disabling eager execution

For TensorFlow 1.x, not enabling eager execution will let the system optimize the program flow as a graph and run it faster.

Use `tf.function`

In TensorFlow 2.x, you cannot disable eager execution (there is a compatibility API, but we shouldn't be using that for anything other than migration from TensorFlow 1.x). Instead, any function that could benefit from a speedup by executing in graph mode can simply be annotated with `@tf.function`. It's worth noting that any function that is called within an annotated function will also run in graph mode. This gives us the advantage of speedup from graph-based execution without sacrificing the debugging capabilities of eager execution. Typically, the best speedup is observed on short computationally intensive tasks:

```
conv_layer = tf.keras.layers.Conv2D(224, 3)

def non_tf_func(image):
  for _ in range(1,3):
        conv_layer(image)
  return
```

```python
@tf.function
def tf_func(image):
  for _ in range(1,3):
        conv_layer(image)
  return

mat = tf.zeros([1, 100, 100, 100])

# Warm up
non_tf_func(mat)
tf_func(mat)

print("Without @tf.function:", timeit.timeit(lambda: non_tf_func(mat),
          number=10000), " seconds")
print("With @tf.function:", timeit.timeit(lambda: tf_func(mat), number=10000),
          "seconds")
```

```
=====Output=====
Without @tf.function: 7.234016112051904 seconds
With @tf.function:    0.7510978290811181 seconds
```

As we can see in our contrived example, simply attributing a function with `@tf.function` has given us a speedup of 10 times, from 7.2 seconds to 0.7 seconds.

```python
@tf.function
def tf_func(image):
  for _ in range(1,3):
        conv_layer(image)
  return

mat = tf.zeros([1, 100, 100, 100])

# Warm up
non_tf_func(mat)
tf_func(mat)

print("Without @tf.function:", timeit.timeit(lambda: non_tf_func(mat),
          number=10000), " seconds")
print("With @tf.function:", timeit.timeit(lambda: tf_func(mat), number=10000),
          "seconds")
```

```
=====Output=====
Without @tf.function: 7.234016112051904 seconds
With @tf.function:    0.7510978290811181 seconds
```

As we can see in our contrived example, simply attributing a function with `@tf.function` has given us a speedup of 10 times, from 7.2 seconds to 0.7 seconds.

Overtrain, and Then Generalize

In machine learning, overtraining on a dataset is considered to be harmful. However, we will demonstrate that we can use overtraining in a controlled fashion to our advantage to make training faster.

As the saying goes, “The perfect is the enemy of the good.” We don’t want our network to be perfect right off the bat. In fact, we wouldn’t even want it to be any good initially. What we really want instead is for it to be learning *something* quickly, even if imperfectly. Because then we have a good baseline that we can fine tune to its highest potential. And experiments have shown that we can get to the end of the journey faster than training conventionally.

NOTE

To further clarify the idea of overtraining and then generalizing, let’s look at an imperfect analogy of language learning. Suppose that you want to learn French. One way is to throw a book of vocabulary and grammar at you and expect you to memorize everything. Sure, you might go through the book every day and maybe in a few years, you might be able to speak some French. But this would not be the optimal way to learn.

Alternatively, we could look at how language learning programs approach this process. These programs introduce you to only a small set of words and grammatical rules initially. After you have learned them, you will be able to speak some broken French. Maybe you could ask for a cup of coffee at a restaurant or ask for directions at a bus stop. At this point, you will be introduced constantly to a larger set of words and rules, and this will help you to improve over time.

This process is similar to how our model would learn gradually with more and more data.

How do we force a network to learn quickly and imperfectly? Make it overtrain on our data. The following three strategies can help.

Use progressive sampling

One approach to overtrain and then generalize is to progressively show more and more of the original training set to the model. Here’s a simple implementation:

1. Take a sample of the dataset (say, roughly 10%).
2. Train the network until it converges; in other words, until it begins to

perform well on the training set.

3. Train on a larger sample (or even the entire training set).

By repeatedly showing a smaller sample of the dataset, the network will learn features much more quickly, but only related to the sample shown. Hence, it would tend to overtrain, usually performing better on the training set compared to the test set. When that happens, exposing the training process to the entire dataset will tend to generalize its learning, and eventually the test set performance would increase.

Use progressive augmentation

Another approach is to train on the entire dataset with little to no data augmentation at first, and then progressively increase the degree of augmentation.

By showing the unaugmented images repeatedly, the network would learn patterns faster, and by progressively increasing the degree of augmentation, it would become more robust.

Use progressive resizing

Another approach, made famous by Jeremy Howard from fast.ai (which offers free courses on AI), is progressive resizing. The key idea behind this approach is to train first on images scaled down to smaller pixel size, and then progressively fine tune on larger and larger sizes until the original image size is reached.

Images resized by half along both the width and height have a 75% reduction in pixels, and theoretically could lead to an increase in training speed of four times over the original images. Similarly, resizing to a quarter of the original height and width can in the best case lead to 16-times reduction (at a lower accuracy). Smaller images have fewer details visible, forcing the network to instead learn higher-level features including broad shapes and colors. Then, training with larger images will help the network learn the finer details, progressively increasing the test accuracy, as well. Just like a child is taught the high-level concepts first and then progressively exposed to more details in later years, the same concept is applied here to CNNs.

TIP

You can experiment with a combination of any of these methods or even build your own creative methods such as training on a subset of classes and then generalizing to all the classes later.

TIP

You can experiment with a combination of any of these methods or even build your own creative methods such as training on a subset of classes and then generalizing to all the classes later.

Install an Optimized Stack for the Hardware

Hosted binaries for open source packages are usually built to run on a variety of hardware and software configurations. These packages try to appeal to the least common denominator. When we do `pip install` on a package, we end up downloading and installing this general-purpose, works-for-everyone binary. This convenience comes at the expense of not being able to take advantage of the specific features offered by a particular hardware stack. This issue is one of the big reasons to avoid installing prebuilt binaries and instead opt for building packages from source.

As an example, Google has a single TensorFlow package on `pip` that can run on an old Sandy Bridge (second-generation Core i3) laptop as well as a powerful 16-core Intel Xeon server. Although convenient, the downside of this is that this package does not take advantage of the highly powerful hardware of the Xeon server. Hence, for CPU-based training and inference, Google recommends compiling TensorFlow from source to best optimize for the hardware at hand.

One way to do this manually is by setting the configuration flags for the hardware before building the source code. For example, to enable support for AVX2 and SSE 4.2 instruction sets, we can simply execute the following build command (note the extra `m` character ahead of each instruction set in the command):

```
$ bazel build -c opt --copt=-mavx2 --copt=-msse4.2
//tensorflow/tools/pip_package:build_pip_package
```

How do you check which CPU features are available? Use the following command (Linux only):

```
$ lscpu | grep Flags

Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid
aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16
xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
cpuid_fault epb cat_l3 cdp_l3 invpcid_single pti intel_ppin ssbd ibrs ibpb stibp
```

```
tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2
erms invpcid rtm cqm rdt_a rdseed adx smap intel_pt xsaveopt cqm_llc
cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts md_clear
flush_l1d
```

Building TensorFlow from source with the appropriate instruction set specified as build flags should result in a substantial increase in speed. The downside here is that building from source can take quite some time, at least a couple of hours. Alternatively, we can use Anaconda to download and install a highly optimized variant of TensorFlow, built by Intel on top of their Math Kernel Library for Deep Neural Networks (MKL-DNN). The installation process is pretty straightforward. First, we install the Anaconda package manager. Then, we run the following command:

```
# For Linux and Mac
$ conda install tensorflow

# For Windows
$ conda install tensorflow-mkl
```

On Xeon CPUs, MKL-DNN often provides upward of two-times speedup in inference.

How about optimization for GPUs? Because NVIDIA abstracts away the differences between the various GPU internals with the CUDA library, there is usually no need to build from source. Instead, we could simply install a GPU variant of TensorFlow from `pip` (`tensorflow-gpu` package). We recommend the Lambda Stack one-liner installer for convenience (along with NVIDIA drivers, CUDA, and cuDNN).

For training and inference on the cloud, AWS, Microsoft Azure, and GCP all provide GPU machine images of TensorFlow optimized for their hardware. It's quick to spin up multiple instances and get started. Additionally, NVIDIA offers GPU-accelerated containers for on-premises and cloud setups.

```
tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2
erms invpcid rtm cqm rdt_a rdseed adx smap intel_pt xsaveopt cqm_llc
cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts md_clear
flush_l1d
```

Building TensorFlow from source with the appropriate instruction set specified as build flags should result in a substantial increase in speed. The downside here is that building from source can take quite some time, at least a couple of hours. Alternatively, we can use Anaconda to download and install a highly optimized variant of TensorFlow, built by Intel on top of their Math Kernel Library for Deep Neural Networks (MKL-DNN). The installation process is pretty straightforward. First, we install the Anaconda package manager. Then, we run the following command:

```
# For Linux and Mac
$ conda install tensorflow

# For Windows
$ conda install tensorflow-mkl
```

On Xeon CPUs, MKL-DNN often provides upward of two-times speedup in inference.

How about optimization for GPUs? Because NVIDIA abstracts away the differences between the various GPU internals with the CUDA library, there is usually no need to build from source. Instead, we could simply install a GPU variant of TensorFlow from `pip` (`tensorflow-gpu` package). We recommend the Lambda Stack one-liner installer for convenience (along with NVIDIA drivers, CUDA, and cuDNN).

For training and inference on the cloud, AWS, Microsoft Azure, and GCP all provide GPU machine images of TensorFlow optimized for their hardware. It's quick to spin up multiple instances and get started. Additionally, NVIDIA offers GPU-accelerated containers for on-premises and cloud setups.

Optimize the Number of Parallel CPU Threads

Compare the following two examples:

```
# Example 1
X = tf.multiply(A, B)
Y = tf.multiply(C, D)

# Example 2
X = tf.multiply(A, B)
Y = tf.multiply(X, C)
```

There are a couple of areas in these examples where we can exploit inherent parallelism:

Between operations

In example 1, the calculation of Y does not depend on the calculation of X. This is because there is no shared data between those two operations, and thus both of them can execute in parallel on two separate threads.

In contrast, in example 2, the calculation of Y depends on the outcome of the first operation (X), and so the second statement cannot execute until the first statement completes execution.

The configuration for the maximum number of threads that can be used for interoperation parallelism is set using the following statement:

```
tf.config.threading.set_inter_op_parallelism_threads(num_threads)
```

The recommended number of threads is equal to the number of CPUs (sockets) on the machine. This value can be obtained by using the `lscpu` command (Linux only).

Per-operation level

We can also exploit the parallelism within a single operation. Operations such as matrix multiplications are inherently parallelizable.

Figure 6-8 demonstrates a simple matrix multiplication operation. It's clear that the overall product can be split into four independent calculations. After

all, the product between one row of a matrix and one column of another matrix does not depend on the calculations for the other rows and columns. Each of those splits could potentially get its own thread and all four of them could execute at the same time.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Figure 6-8. A matrix multiplication for A x B operation with one of the multiplications highlighted

The configuration for the number of threads that can be used for intraoperation parallelism is set using the following statement:

```
tf.config.threading.set_intra_op_parallelism_threads(num_threads)
```

The recommended number of threads is equal to the number of cores per CPU. You can obtain this value by using the `lscpu` command on Linux.

all, the product between one row of a matrix and one column of another matrix does not depend on the calculations for the other rows and columns. Each of those splits could potentially get its own thread and all four of them could execute at the same time.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Figure 6-8. A matrix multiplication for A x B operation with one of the multiplications highlighted

The configuration for the number of threads that can be used for intraoperation parallelism is set using the following statement:

```
tf.config.threading.set_intra_op_parallelism_threads(num_threads)
```

The recommended number of threads is equal to the number of cores per CPU. You can obtain this value by using the `lscpu` command on Linux.

Use Better Hardware

If you have already maximized performance optimizations and still need faster training, you might be ready for some new hardware. Replacing spinning hard drives with SSDs can go a long way, as can adding one or more better GPUs. And let’s not forget, sometimes the CPU can be the culprit.

In fact, you might not need to spend much money: public clouds like AWS, Azure, and GCP all provide the ability to rent powerful configurations for a few dollars per hour. Best of all, they come with optimized TensorFlow stacks preinstalled.

Of course, if you have the cash to spend or have a rather generous expense account, you could just skip this entire chapter and buy the 2-petaFLOPS NVIDIA DGX-2. Weighing in at 163 kgs (360 pounds), its 16 V100 GPUs (with a total of 81,920 CUDA cores) consume 10 kW of power—the equivalent of seven large window air conditioners. And all it costs is $400,000!

Figure 6-9. The $400,000 NVIDIA DGX-2 deep learning system

Use Better Hardware

If you have already maximized performance optimizations and still need faster training, you might be ready for some new hardware. Replacing spinning hard drives with SSDs can go a long way, as can adding one or more better GPUs. And let's not forget, sometimes the CPU can be the culprit.

In fact, you might not need to spend much money: public clouds like AWS, Azure, and GCP all provide the ability to rent powerful configurations for a few dollars per hour. Best of all, they come with optimized TensorFlow stacks preinstalled.

Of course, if you have the cash to spend or have a rather generous expense account, you could just skip this entire chapter and buy the 2-petaFLOPS NVIDIA DGX-2. Weighing in at 163 kgs (360 pounds), its 16 V100 GPUs (with a total of 81,920 CUDA cores) consume 10 kW of power—the equivalent of seven large window air conditioners. And all it costs is $400,000!

Figure 6-9. The $400,000 NVIDIA DGX-2 deep learning system

Distribute Training

"*Two lines to scale training horizontally!*"

On a single machine with a single GPU, there's only so far that we can go. Even the beefiest GPUs have an upper limit in compute power. Vertical scaling can take us only so far. Instead, we look to scale horizontally—distribute computation across processors. We can do this across multiple GPUs, TPUs, or even multiple machines. In fact, that is exactly what researchers at Google Brain did back in 2012, using 16,000 processors to run a neural network built to look at cats on YouTube.

In the dark days of the early 2010s, training on ImageNet used to take anywhere from several weeks to months. Multiple GPUs would speed things up, but few people had the technical know-how to configure such a setup. It was practically out of reach for beginners. Luckily, we live in the day of TensorFlow 2.0, in which setting up distributed training is a matter of introducing two lines of code:

```
mirrored_strategy = tf.distribute.MirroredStrategy()
with mirrored_strategy.scope():
  model = tf.keras.applications.ResNet50()
  model.compile(loss="mse", optimizer="sgd")
```

Training speed increases nearly proportionally (90–95%) in relation to the number of GPUs added. As an example, if we added four GPUs (of similar compute power), we would notice an increase of >3.6 times speedup ideally.

Still, a single system can only support a limited number of GPUs. How about multiple nodes, each with multiple GPUs? Similar to `MirroredStrategy`, we can use `MultiWorkerMirroredStrategy`. This is quite useful when building a cluster on the cloud. Table 6-1 presents a couple of distribution strategies for different use cases.

Table 6-1. Recommended distribution strategies

Strategy	Use case
`MirroredStrategy`	Single node with two or more GPUs
	Multiple nodes with one or more GPUs each

```
MultiWorkerMirroredStrategy
```

To get the cluster nodes to communicate with one another for `MultiWorkerMirroredStrategy`, we need to configure the `TF_CONFIG` environment variable on every single host. This requires setting up a JSON object that contains the IP addresses and ports of all other hosts in the cluster. Manually managing this can be error prone, and this is where orchestration frameworks like Kubernetes really shine.

NOTE

The open source Horovod library from Uber is another high-performance and easy-to-use distribution framework. Many of the record benchmark performances seen in the next section require distributed training on several nodes, and Horovod's performance helped them get the edge. It is worth noting that the majority of the industry uses Horovod particularly because distributed training on earlier versions of TensorFlow was a much more involved process. Additionally, Horovod works with all major deep learning libraries with minimal amount of code change or expertise. Often configured through the command line, running a distributed program on four nodes, each with four GPUs, can be done in a single command line:

```
$ horovodrun -np 16 -H
server1:4,server2:4,server3:4,server4:4 python
train.py
```

```
MultiWorkerMirroredStrategy
```

To get the cluster nodes to communicate with one another for `MultiWorkerMirroredStrategy`, we need to configure the `TF_CONFIG` environment variable on every single host. This requires setting up a JSON object that contains the IP addresses and ports of all other hosts in the cluster. Manually managing this can be error prone, and this is where orchestration frameworks like Kubernetes really shine.

NOTE

The open source Horovod library from Uber is another high-performance and easy-to-use distribution framework. Many of the record benchmark performances seen in the next section require distributed training on several nodes, and Horovod's performance helped them get the edge. It is worth noting that the majority of the industry uses Horovod particularly because distributed training on earlier versions of TensorFlow was a much more involved process. Additionally, Horovod works with all major deep learning libraries with minimal amount of code change or expertise. Often configured through the command line, running a distributed program on four nodes, each with four GPUs, can be done in a single command line:

```
$ horovodrun -np 16 -H
server1:4,server2:4,server3:4,server4:4 python
train.py
```

Examine Industry Benchmarks

Three things were universally popular in the 1980s—long hair, the Walkman, and database benchmarks. Much like the current hype of deep learning, database software was similarly going through a phase of making bold promises, some of which were marketing hype. To put these companies to the test, a few benchmarks were introduced, more famously among them was the Transaction Processing Council (TPC) benchmark. When someone needed to buy database software, they could rely on this public benchmark to decide where to spend their company's budget. This competition fueled rapid innovation, increasing speed and performance per dollar, moving the industry ahead faster than anticipated.

Inspired by TPC and other benchmarks, a few system benchmarks were created to standardize performance reporting in machine learning.

DAWNBench

> Stanford's DAWNBench benchmarks time and cost to get a model to 93% Top-5 accuracy on ImageNet. Additionally, it also does a time and cost leaderboard on inference time. It's worth appreciating the rapid pace of performance improvement for training such a massive network. When DAWNBench originally started in September 2017, the reference entry trained in 13 days at a cost of $2,323.39. In just one and a half years since then, although the cheapest training costs as low as $12, the fastest training time is 2 minutes 43 seconds. Best of all, most entries contain the training source code and optimizations that can be studied and replicated by us. This gives further guidance on the effects of hyperparameters and how we can use the cloud for cheap and fast training without breaking the bank.

Table 6-2. Entries on DAWNBench as of August 2019, sorted by the least cost for training a model to 93% Top-5 accuracy

Cost (USD)	Training time	Model	Hardware	Framework
$12.60	2:44:31	ResNet-50 Google Cloud TPU	GCP n1-standard-2, Cloud TPU	TensorFlow 1.11

$20.89	1:42:23	ResNet-50 Setu Chokshi (MS AI MVP)	Azure ND40s_v2	PyTorch 1.0
$42.66	1:44:34	ResNet-50 v1 GE Healthcare (Min Zhang)	8*V100 (single p3.16x large)	TensorFlow 1.11 + Horovod
$48.48	0:29:43	ResNet-50 Andrew Shaw, Yaroslav Bulatov, Jeremy Howard	32 * V100 (4x - AWS p3.16x large)	Ncluster + PyTorch 0.5

MLPerf

Similar to DAWNBench, MLPerf is aimed at repeatable and fair testing of AI system performance. Although newer than DAWNBench, this is an industry consortium with much wider support, especially on the hardware side. It runs challenges for both training and inference in two divisions: open and closed. The closed division trains the same model with the same optimizers, so the raw hardware performance can be compared apples-to-apples. The open division, on the other hand, allows using faster models and optimizers to allow for more rapid progress. Compared to the more cost-effective entries in DAWNBench in Table 6-2, the top performers on MLPerf as shown in Table 6-3 might be a bit out of reach for most of us. The top-performing NVIDIA DGX SuperPod, composed of 96 DGX-2H with a total of 1,536 V100 GPUs, costs in the $35 to $40 million range. Even though 1,024 Google TPUs might themselves cost in the several millions, they are each available to rent on the cloud at $8/hour on-demand pricing (as of August 2019), resulting in a net cost of under $275 for the less-than two minutes of training time.

Table 6-3. Key closed-division entries on DAWNBench as of August 2019, showing training time for a ResNet-50 model to get to 75.9% Top-1 accuracy

Time (minutes)	Submitter	Hardware	Accelerator	# of accelerators
1.28	Google	TPUv3	TPUv3	1,024

1.33	NVIDIA	96x DGX-2H	Tesla V100	1,536
8,831.3	Reference	Pascal P100	Pascal P100	1

Although both the aforementioned benchmarks highlight training as well as inference (usually on more powerful devices), there are other inference-specific competitions on low-power devices, with the aim to maximize accuracy and speed while reducing power consumption. Held at annual conferences, here are some of these competitions:

- LPIRC: Low-Power Image Recognition Challenge
- EDLDC: Embedded Deep Learning Design Contest
- System Design Contest at Design Automation Conference (DAC)

1.33	NVIDIA	96x DGX-2H	Tesla V100	1,536
8,831.3	Reference	Pascal P100	Pascal P100	1

Although both the aforementioned benchmarks highlight training as well as inference (usually on more powerful devices), there are other inference-specific competitions on low-power devices, with the aim to maximize accuracy and speed while reducing power consumption. Held at annual conferences, here are some of these competitions:

- LPIRC: Low-Power Image Recognition Challenge
- EDLDC: Embedded Deep Learning Design Contest
- System Design Contest at Design Automation Conference (DAC)

Inference

Training our model is only half the game. We eventually need to serve the predictions to our users. The following points guide you to making your serving side more performant.

Inference

Training our model is only half the game. We eventually need to serve the predictions to our users. The following points guide you to making your serving side more performant.

Use an Efficient Model

Deep learning competitions have traditionally been a race to come up with the highest accuracy model, get on top of the leaderboard, and get the bragging rights. But practitioners live in a different world—the world of serving their users quickly and efficiently. With devices like smartphones, edge devices, and servers with thousands of calls per second, being efficient on all fronts (model size and computation) is critically needed. After all, many machines would not be capable of serving a half gigabyte VGG-16 model, which happens to need 30 billion operations to execute, for not even that high of accuracy. Among the wide variety of pretrained architectures available, some are on the higher end of accuracy but large and resource intensive, whereas others provide modest accuracy but are much lighter. Our goal is to pick the architecture that can deliver the highest accuracy for the available computational power and memory budget of our inference device. In Figure 6-10, we want to pick models in the upper-left zone.

Figure 6-10. Comparing different models for size, accuracy, and operations per second (adapted from "Ar

Analysis of Deep Neural Network Models for Practical Applications" by Alfredo Canziani, Adam Paszke, and Eugenio Culurciello)

Usually, the approximately 15 MB MobileNet family is the go-to model for efficient smartphone runtimes, with more recent versions like MobileNetV2 and MobileNetV3 being better than their predecessors. Additionally, by varying the hyperparameters of the MobileNet models like depth multiplier, the number of computations can be further reduced, making it ideal for real-time applications. Since 2017, the task of generating the most optimal architecture to maximize accuracy has also been automated with NAS. It has helped discover new (rather obfuscated looking) architectures that have broken the ImageNet accuracy metric multiple times. For example, FixResNeXt (based on PNASNet architecture at 829 MB) reaches a whopping 86.4% Top-1 accuracy on ImageNet. So, it was natural for the research community to ask whether NAS helps find architecture that's tuned for mobile, maximizing accuracy while minimizing computations. The answer is a resounding yes—resulting in faster and better models, optimized for the hardware at hand. As an example, MixNet (July 2019) outperforms many state-of-the-art models. Note how we went from billions of floating-point operations to millions (Figure 6-10 and Figure 6-11).

Figure 6-11. Comparison of several mobile-friendly models in the paper "MixNet: Mixed Depthwise Convolution Kernels" by Mingxing Tan and Quoc V. Le

As practitioners, where can we find current state-of-the-art models? *PapersWithCode.com/SOTA* showcases leaderboards on several AI problems, comparing paper results over time, along with the model code. Of particular interest would be the models with a low number of parameters that achieve high accuracies. For example, EfficientNet gets an amazing Top-1 84.4% accuracy with 66 million parameters, so it could be an ideal candidate for running on servers. Additionally, the ImageNet test metrics are on 1,000 classes, whereas our case might just require classification on a few classes. For those cases, a much smaller model would suffice. Models listed in Keras Application (*tf.keras.applications*), TensorFlow Hub, and TensorFlow Models usually carry many variations (input image sizes, depth multipliers, quantizations, etc.).

TIP

Shortly after Google AI researchers publish a paper, they release the model used in the paper on the TensorFlow Models repository.

As practitioners, where can we find current state-of-the-art models? *PapersWithCode.com/SOTA* showcases leaderboards on several AI problems, comparing paper results over time, along with the model code. Of particular interest would be the models with a low number of parameters that achieve high accuracies. For example, EfficientNet gets an amazing Top-1 84.4% accuracy with 66 million parameters, so it could be an ideal candidate for running on servers. Additionally, the ImageNet test metrics are on 1,000 classes, whereas our case might just require classification on a few classes. For those cases, a much smaller model would suffice. Models listed in Keras Application (*tf.keras.applications*), TensorFlow Hub, and TensorFlow Models usually carry many variations (input image sizes, depth multipliers, quantizations, etc.).

TIP

Shortly after Google AI researchers publish a paper, they release the model used in the paper on the TensorFlow Models repository.

Quantize the Model

"*Represent 32-bit weights to 8-bit integer, get 2x faster, 4x smaller models*"

Neural networks are driven primarily by matrix–matrix multiplications. The arithmetic involved tends to be rather forgiving in that small deviations in values do not cause a significant swing in output. This makes neural networks fairly robust to noise. After all, we want to be able to recognize an apple in a picture, even in less-than-perfect lighting. When we quantize, we essentially take advantage of this "forgiving" nature of neural networks.

Before we look at the different quantization techniques, let's first try to build an intuition for it. To illustrate quantized representations with a simple example, we'll convert 32-bit floating-point weights to INT8 (8-bit integer) using *linear quantization*. Obviously, FP32 represents 2^{32} values (hence, 4 bytes to store), whereas INT8 represents $2^8 = 256$ values (1 byte). To quantize:

1. Find the minimum and maximum values represented by FP32 weights in the neural network.
2. Divide this range into 256 intervals, each corresponding to an INT8 value.
3. Calculate a scaling factor that converts an INT8 (integer) back to a FP32. For example, if our original range is from 0 to 1, and INT8 numbers are 0 to 255, the scaling factor will be 1/256.
4. Replace FP32 numbers in each interval with the INT8 value. Additionally, store the scaling factor for the inference stage where we convert INT8 values back to FP32 values. This scaling factor only needs to be stored once for the entire group of quantized values.
5. During inference calculations, multiply the INT8 values by the scaling factor to convert it back to a floating-point representation. Figure 6-12 illustrates an example of linear quantization for the interval [0, 1].

Figure 6-12. Quantizing from a 0 to 1 32-bit floating-point range down to an 8-bit integer range for reduced storage space

There are a few different ways to quantize our models, the simplest one being reducing the bit representation of the weights from 32-bit to 16-bit or lower. As might be evident, converting 32-bit to 16-bit means half the memory size is needed to store a model. Similarly, converting to 8-bit would require a quarter of the size. So why not convert it to 1-bit and save 32x the size? Well, although the models are forgiving up to a certain extent, with each reduction, we will notice a drop in accuracy. This reduction in accuracy grows exponentially beyond a certain threshold (especially below 8 bits). To go below and still have a useful working model (like a 1-bit representation), we'd need to follow a special conversion process to convert them to binarized neural networks. XNOR.ai, a deep learning startup, has famously been able to bring this technique to production. The Microsoft Embedded Learning Library (ELL) similarly provides such tools, which have a lot of value for edge devices like the Raspberry Pi.

There are numerous benefits to quantization:

Improved memory usage

By quantizing to 8-bit integer representation (INT8), we typically get a 75% reduction in model size. This makes it more convenient to store and load the model in memory.

Improved performance

Integer operations are faster than floating-point operations. Additionally, the savings in memory usage reduces the likelihood of having to unload the model from RAM during execution, which also has the added benefit of decreased power consumption.

Portability

Edge devices such as Internet of Things devices might not support floating-point arithmetic, so it would be untenable to keep the model as a floating-point in such situations.

Most inference frameworks provide a way to quantize, including Core ML Tools from Apple, TensorRT from NVIDIA (for servers), and TensorFlow Lite, as well as the TensorFlow Model Optimization Toolkit from Google. With TensorFlow

Lite, models can be quantized after training during conversion (called post-training quantization). To minimize accuracy losses even further, we can use the TensorFlow Model Optimization Toolkit during training. This process is called *quantization-aware training*.

It would be useful to measure the benefit provided by quantization. Metrics from the TensorFlow Lite Model optimization benchmarks (shown in Table 6-4) give us a hint, comparing 1) unquantized, 2) post-training quantized, and 3) quantization-aware trained models. The performance was measured on a Google Pixel 2 device.

Table 6-4. Effects of different quantization strategies (8-bit) on models (source: TensorFlow Lite model optimization documentation)

Model		MobileNet	MobileNetV2	InceptionV3
Top-1 accuracy	Original	0.709	0.719	0.78
	Post-training quantized	0.657	0.637	0.772
	Quantization-aware training	0.7	0.709	0.775
Latency (ms)	Original	124	89	1130
	Post-training quantized	112	98	845
	Quantization-aware training	64	54	543
Size (MB)	Original	16.9	14	95.7
	Optimized	4.3	3.6	23.9

So, what do these numbers indicate? After quantization using TensorFlow Lite to INT8, we see roughly a four-times reduction in size, approximately two-times speedup in run time, and less than 1% change in accuracy. Not bad!

More extreme form of quantization, like 1-bit binarized neural networks (like XNOR-Net), claim a whopping 58-times speedup with roughly 32-times smaller size when tested on AlexNet, with a 22% loss in accuracy.

Prune the Model

Pick a number. Multiply it by 0. What do we get? Zero. Multiply your pick again by a small value neighboring 0, like 10^{-6}, and we'll still get an insignificant value. If we replace such tiny weights ($\rightarrow 0$) in a model with 0 itself, it should have little effect on the model's predictions. This is called *magnitude-based weight pruning*, or simply pruning, and is a form of *model compression*. Logically, putting a weight of 0 between two nodes in a fully connected layer is equivalent to deleting the edge between them. This makes a model with dense connections sparser.

As it happens, a large chunk of the weights in a model are close to 0. Pruning the model will result in many of those weights being set to 0. This happens with little impact to accuracy. Although this does not save any space by itself, it introduces a ton of redundancy that can be exploited when it comes time to save the model to disk in a compressed format such as ZIP. (It is worth noting that compression algorithms thrive on repeating patterns. The more the repetition, the higher the compressibility.) The end result is that our model can often be compressed by four times. Of course, when we finally need to use the model, it would need to be uncompressed before loading in memory for inference.

The TensorFlow team observed the accuracy loss shown in Table 6-5 while pruning the models. As expected, more efficient models like MobileNet observe higher (though still small) accuracy loss when compared with comparatively bigger models like InceptionV3.

Table 6-5. Model accuracy loss versus pruning percentage

Model	Sparsity	Accuracy loss against original accuracy
InceptionV3	50%	0.1%
InceptionV3	75%	2.5%
InceptionV3	87.5%	4.5%
MobileNet	50%	2%

Keras provides APIs to prune our model. This process can be done iteratively

during training. Train a model normally or pick a pretrained model. Then, periodically prune the model and continue training. Having enough epochs between the periodic prunes allows the model to recover from any damage due to introducing so much sparsity. The amount of sparsity and number of epochs between prunes can be treated as hyperparameters to be tuned.

Another way of implementing this is by using Tencent's PocketFlow tool, a one-line command that provides several other pruning strategies implemented in recent research papers.

during training. Train a model normally or pick a pretrained model. Then, periodically prune the model and continue training. Having enough epochs between the periodic prunes allows the model to recover from any damage due to introducing so much sparsity. The amount of sparsity and number of epochs between prunes can be treated as hyperparameters to be tuned.

Another way of implementing this is by using Tencent's PocketFlow tool, a one-line command that provides several other pruning strategies implemented in recent research papers.

Use Fused Operations

In any serious CNN, the convolutional layer and batch normalization layer frequently appear together. They are kind of the Laurel and Hardy of CNN layers. Fundamentally, they are both linear operations. Basic linear algebra tells us that combining two or more linear operations will also result in a linear operation. By combining convolutional and batch normalization layers, we not only reduce the number of computations, but also decrease the amount of time spent in data transfer, both between main memory and GPU, and main memory and CPU registers/cache. Making them one operation prevents an extra roundtrip. Luckily, for inference purposes, most inference frameworks either automatically do this fusing step or provide model converters (like TensorFlow Lite) to make this optimization while converting the model to the inference format.

Use Fused Operations

In any serious CNN, the convolutional layer and batch normalization layer frequently appear together. They are kind of the Laurel and Hardy of CNN layers. Fundamentally, they are both linear operations. Basic linear algebra tells us that combining two or more linear operations will also result in a linear operation. By combining convolutional and batch normalization layers, we not only reduce the number of computations, but also decrease the amount of time spent in data transfer, both between main memory and GPU, and main memory and CPU registers/cache. Making them one operation prevents an extra roundtrip. Luckily, for inference purposes, most inference frameworks either automatically do this fusing step or provide model converters (like TensorFlow Lite) to make this optimization while converting the model to the inference format.

Enable GPU Persistence

Loading and initializing the GPU drivers take time. You might have noticed a delay every time a training or inference job was initiated. For frequent, short jobs, the overhead can become relatively expensive quickly. Imagine an image classification program for which the classification takes 10 seconds, 9.9 of which were spent in loading the driver. What we need is for the GPU driver to stay preinitialized in the background, and be ready for whenever our training jobs start. And that's where the NVIDIA GPU Persistence Daemon comes to the rescue:

```
$ nvidia-persistenced --user {YOUR_USERNAME}
```

Our GPUs will use a bit more wattage during idle time, but they will be ready and available the next time a program is launched.

Enable GPU Persistence

Loading and initializing the GPU drivers take time. You might have noticed a delay every time a training or inference job was initiated. For frequent, short jobs, the overhead can become relatively expensive quickly. Imagine an image classification program for which the classification takes 10 seconds, 9.9 of which were spent in loading the driver. What we need is for the GPU driver to stay preinitialized in the background, and be ready for whenever our training jobs start. And that's where the NVIDIA GPU Persistence Daemon comes to the rescue:

```
$ nvidia-persistenced --user {YOUR_USERNAME}
```

Our GPUs will use a bit more wattage during idle time, but they will be ready and available the next time a program is launched.

Summary

In this chapter, we explored different avenues for improving the speed and performance of our deep learning pipeline, from storing and reading the data to inference. A slow data pipeline often leads to a GPU starving for data, resulting in idle cycles. With several of the simple optimizations we discussed, our hardware can be put to its maximum efficiency. The handy checklist can serve as a ready reference. Feel free to make a copy for your desk (or your refrigerator). With these learnings, we hope to see your entry among the top performers of the MLPerf benchmark list.

Summary

In this chapter, we explored different avenues for improving the speed and performance of our deep learning pipeline, from storing and reading the data to inference. A slow data pipeline often leads to a GPU starving for data, resulting in idle cycles. With several of the simple optimizations we discussed, our hardware can be put to its maximum efficiency. The handy checklist can serve as a ready reference. Feel free to make a copy for your desk (or your refrigerator). With these learnings, we hope to see your entry among the top performers of the MLPerf benchmark list.

Chapter 7. Practical Tools, Tips, and Tricks

This chapter contains material that we, your authors, have encountered during our professional work as well as while working on this book, primarily during experimentation. The material covered here doesn't necessarily fit in any single chapter; rather, it's material that deep learning practitioners could find useful on a day-to-day basis across a variety of tasks. In line with the "practical" theme, these questions cover a range of helpful pragmatic guidelines across topics including setting up an environment, training, model interoperability, data collection and labeling, code quality, managing experiments, team collaboration practices, privacy, and further exploration topics.

Due to the fast-changing pace of the AI field, this chapter is a small subset of the "living" document hosted on the book's Github repository (see *http://PracticalDeepLearning.ai*) at *code/chapter-9*, where it is constantly evolving. If you have more questions or, even better, answers that might help other readers, feel free to tweet them @PracticalDLBook or submit a pull request.

Chapter 7. Practical Tools, Tips, and Tricks

This chapter contains material that we, your authors, have encountered during our professional work as well as while working on this book, primarily during experimentation. The material covered here doesn't necessarily fit in any single chapter; rather, it's material that deep learning practitioners could find useful on a day-to-day basis across a variety of tasks. In line with the "practical" theme, these questions cover a range of helpful pragmatic guidelines across topics including setting up an environment, training, model interoperability, data collection and labeling, code quality, managing experiments, team collaboration practices, privacy, and further exploration topics.

Due to the fast-changing pace of the AI field, this chapter is a small subset of the "living" document hosted on the book's Github repository (see *http://PracticalDeepLearning.ai*) at *code/chapter-9*, where it is constantly evolving. If you have more questions or, even better, answers that might help other readers, feel free to tweet them @PracticalDLBook or submit a pull request.

Installation

Q: *I came across an interesting and useful Jupyter Notebook on GitHub. Making the code run will require cloning the repository, installing packages, setting up the environment, and more steps. Is there an instant way to run it interactively?*

Simply enter the Git repository URL into Binder (*mybinder.org*), which will turn it into a collection of interactive notebooks. Under the hood, it will search for a dependency file, like *requirements.txt* or *environment.yml* in the repository's root directory. This will be used to build a Docker image, to help run the notebook interactively in your browser.

Q: *What is the quickest way to get my deep learning setup running on a fresh Ubuntu machine with NVIDIA GPUs?*

Life would be great if `pip install tensorflow-gpu` would solve everything. However, that's far from reality. On a freshly installed Ubuntu machine, listing all the installation steps would take at least three pages and more than an hour to follow, including installing NVIDIA GPU drivers, CUDA, cuDNN, Python, TensorFlow, and other packages. And then it requires carefully checking the version interoperability between CUDA, cuDNN and TensorFlow. More often than not, this ends in a broken system. A world of pain to say the least!

Wouldn't it be great if two lines could solve all of this effortlessly? Ask, and ye shall receive:

```
$ sudo apt update && sudo ubuntu-drivers autoinstall && sudo reboot
$ export LAMBDA_REPO=$(mktemp) \
&& wget -O${LAMBDA_REPO} \
https://lambdalabs.com/static/misc/lambda-stack-repo.deb \
&& sudo dpkg -i ${LAMBDA_REPO} && rm -f ${LAMBDA_REPO} \
&& sudo apt-get update && sudo apt-get install -y lambda-stack-cuda \
&& sudo reboot
```

The first line ensures that all the drivers are updated. The second line is brought to us by the Lambda Labs, a San Francisco–based deep learning hardware and cloud provider. The command sets up the Lambda Stack, which installs TensorFlow, Keras, PyTorch, Caffe, Caffe2, Theano, CUDA, cuDNN, and

NVIDIA GPU drivers. Because the company needs to install the same deep learning packages on thousands of machines, it automated the process with a one-line command and then open sourced it so that others can also make use of it.

Q: *What is the fastest way to install TensorFlow on a Windows PC?*

1. Install Anaconda Python 3.7.
2. On the command line, run `conda install tensorflow-gpu`.
3. If you do not have GPUs, run `conda install tensorflow`.

One additional benefit of a CPU-based Conda installation is that it installs Intel MKL optimized TensorFlow, running faster than the version we get by using `pip install tensorflow`.

Q: *I have an AMD GPU. Could I benefit from GPU speedups in TensorFlow on my existing system?*

Although the majority of the deep learning world uses NVIDIA GPUs, there is a growing community of people running on AMD hardware with the help of the ROCm stack. Installation using the command line is simple:

1. `sudo apt install rocm-libs miopen-hip cxlactivitylogger`
2. `sudo apt install wget python3-pip`
3. `pip3 install --user tensorflow-rocm`

Q: *Forget installation, where can I get preinstalled deep learning containers?*

Docker is synonymous with setting up environments. Docker helps run isolated containers that are bundled with tools, libraries, and configuration files. There are several deep learning Docker containers available while selecting your virtual machine (VM) from major cloud providers AWS, Microsoft Azure, GCP, Alibaba, etc.) that are ready to start working. NVIDIA also freely provides NVIDIA GPU Cloud containers, which are the same high-performance containers used to break training speed records on the MLPerf benchmarks. You can even run these containers on your desktop machine.

)

Training

Q: *I don't like having to stare at my screen constantly to check whether my training finished. Can I get a notification alert on my phone, instead?*

Use Knock Knock, a Python library that, as the name suggests, notifies you when your training ends (or your program crashes) by sending alerts on email, Slack, or even Telegram! Best of all, it requires adding only two lines of code to your training script. No more opening your program a thousand times to check whether the training has finished.

Q: *I prefer graphics and visualizations over plain text. Can I get real-time visualizations for my training process?*

FastProgress progress bar (originally developed for fast.ai by Sylvain Gugger) comes to the rescue.

Q: *I conduct a lot of experiments iteratively and often lose track of what changed between each experiment as well as the effect of the change. How do I manage my experiments in a more organized manner?*

Software development has had the ability to keep a historical log of changes through version control. Machine learning, unfortunately, did not have the same luxury. That's changing now with tools like Weights and Biases, and Comet.ml. They allow you to keep track of multiple runs and to log training curves, hyperparameters, outputs, models, notes, and more with just two lines of code added to your Python script. Best of all, through the power of the cloud, you can conveniently track experiments even if you are away from the machine, and share the results with others.

Q: *How do I check whether TensorFlow is using the GPU(s) on my machine?*

Use the following handy command:

```
tf.test.is_gpu_available()
```

Q: *I have multiple GPUs on my machine. I don't want my training script to consume all of them. How do I restrict my script to run on only a specific GPU?*

Use `CUDA_VISIBLE_DEVICES=GPU_ID`. Simply prefix the training script command as follows:

```
$ CUDA_VISIBLE_DEVICES=GPU_ID python train.py
```

Alternatively, write the following lines early on in your training script:

```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="GPU_ID"
```

`GPU_ID` can have values such as 0, 1, 2, and so on. You can see these IDs (along with GPU usage) using the `nvidia-smi` command. For assigning to multiple GPUs, use a comma-separated list of IDs.

Q: *Sometimes it feels like there are too many knobs to adjust when training. Can it be done automatically, instead, to get the best accuracy?*

There are many options for automated hyperparameter tuning, including Keras-specific Hyperas and Keras Tuner, and more generic frameworks such as Hyperopt and Bayesian optimization that perform extensive experimentation to maximize our objective (i.e., maximizing accuracy in our case) more intelligently than simple grid searches.

Q: *ResNet and MobileNet work well enough for my use case. Is it possible to build a model architecture that can achieve even higher accuracy for my scenario?*

Three words: Neural Architecture Search (NAS). Let the algorithm find the best architecture for you. NAS can be accomplished through packages like Auto-Keras and AdaNet.

Q: *How do I go about debugging my TensorFlow script?*

The answer is in the question: TensorFlow Debugger (`tfdbg`).

Use `CUDA_VISIBLE_DEVICES=GPU_ID`. Simply prefix the training script command as follows:

```
$ CUDA_VISIBLE_DEVICES=GPU_ID python train.py
```

Alternatively, write the following lines early on in your training script:

```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="GPU_ID"
```

`GPU_ID` can have values such as 0, 1, 2, and so on. You can see these IDs (along with GPU usage) using the `nvidia-smi` command. For assigning to multiple GPUs, use a comma-separated list of IDs.

Q: *Sometimes it feels like there are too many knobs to adjust when training. Can it be done automatically, instead, to get the best accuracy?*

There are many options for automated hyperparameter tuning, including Keras-specific Hyperas and Keras Tuner, and more generic frameworks such as Hyperopt and Bayesian optimization that perform extensive experimentation to maximize our objective (i.e., maximizing accuracy in our case) more intelligently than simple grid searches.

Q: *ResNet and MobileNet work well enough for my use case. Is it possible to build a model architecture that can achieve even higher accuracy for my scenario?*

Three words: Neural Architecture Search (NAS). Let the algorithm find the best architecture for you. NAS can be accomplished through packages like Auto-Keras and AdaNet.

Q: *How do I go about debugging my TensorFlow script?*

The answer is in the question: TensorFlow Debugger (`tfdbg`).

Model

Q: *I want to quickly know the input and output layers of my model without writing code. How can I accomplish that?*

Use Netron. It graphically shows your model, and on clicking any layer, provides details on the architecture.

Q: *I need to publish a research paper. Which tool should I use to draw my organic, free-range, gluten-free model architecture?*

MS Paint, obviously! No, we're just kidding. We are fans of NN-SVG as well as PlotNeuralNet for creating high-quality CNN diagrams.

Q: *Is there a one-stop shop for all models?*

Indeed! Explore *PapersWithCode.com*, *ModelZoo.co*, and *ModelDepot.io* for some inspiration.

Q: *I've finished training my model. How can I make it available for others to use?*

You can begin by making the model available for download from GitHub. And then list it on the model zoos mentioned in the previous answer. For even wider adoption, upload it to TensorFlow Hub (*tfhub.dev*).

In addition to the model, you should publish a "model card," which is essentially like a résumé of the model. It's a short report that details author information, accuracy metrics, and the dataset it was benchmarked on. Additionally, it provides guidance on potential biases and out-of-scope uses.

Q: *I have a model previously trained in framework X, but I need to use it in framework Y. Do I need to waste time retraining it in framework Y?*

Nope. All you need is the power of the ONNX. For models not in the TensorFlow ecosystem, most major deep learning libraries support saving them in ONNX format, which can then be converted to the TensorFlow format. Microsoft's MMdnn can help in this conversion.

Data

Q: *Could I collect hundreds of images on a topic in a few minutes?*

Yes, you can collect hundreds of images in three minutes or less with a Chrome extension called Fatkun Batch Download Image. Simply search for a keyword in your favorite image search engine, filter images by the correct usage rights (e.g., Public Domain), and press the Fatkun extension to download all images. See Chapter 12, where we use it to build a Not Hotdog app.

Bonus tip: to download from a single website, search for a keyword followed by site:website_address. For example, "horse site:flickr.com."

Q: *Forget the browser. How do I scrape Google for images using the command line?*

```
$ pip install google_images_download
$ googleimagesdownload -k=horse -l=50 -r=labeled-for-reuse
```

`-k`, `-l`, and `-r` are shorthand for `keyword`, `limit` (number of images), and `usage_rights`, respectively. This is a powerful tool with many options for controlling and filtering what images to download from Google searches. Plus, instead of just loading the thumbnails shown by Google Images, it saves the original images linked by the search engine. For saving more than 100 images, install the `selenium` library along with `chromedriver`.

Q: *Those were not enough for collecting images. I need more control. What other tools can help me download data in more custom ways beyond the search engine?*

With a GUI (no programming needed):

ScrapeStorm.com

Easy GUI to identify rules for elements to extract

WebScraper.io

Chrome-based scraping extension, especially for extracting structured output from single websites

80legs.com

Cloud-based scalable scraper, for parallel, large tasks

Python-based programmatic tools:

Scrapy.org

For more programmable controls on scraping, this is one of the most famous scrapers. Compared to building your own naive scraper to explore websites, it offers throttling rate by domain, proxy, and IP; can handle *robots.txt*; offers flexibility in browser headers to show to web servers; and takes care of several possible edge cases.

InstaLooter

A Python-based tool for scraping Instagram.

Q: *I have the images for the target classes, but now need images for the negative (not item/background) class. Any quick ways to build a big dataset of negative classes?*

ImageN offers 1,000 images—5 random images for 200 ImageNet categories—which you can use as the negative class. If you need more, download a random sample programmatically from ImageNet.

Q: *How can I search for a prebuilt dataset that suits my needs?*

Try Google Dataset Search, *VisualData.io*, and *DatasetList.com*.

Q: *For datasets like ImageNet, downloading, figuring out the format, and then loading them for training takes far too much time. Is there an easy way to read popular datasets?*

TensorFlow Datasets is a growing collection of datasets ready to use with TensorFlow. It includes ImageNet, COCO (37 GB), and Open Images (565 GB) among others. These datasets are exposed as `tf.data.Datasets`, along with performant code to feed them in your training pipeline.

Q: *Training on the millions of ImageNet images will take a long, long time. Is there a smaller representative dataset I could try training on, to quickly experiment and iterate with?*

Try Imagenette. Built by Jeremy Howard from fast.ai, this 1.4 GB dataset contains only 10 classes instead of 1,000.

Q: *What are the largest readily available datasets that I could use for training?*

- Tencent ML Images: 17.7 million images with 11,000 category labels
- Open Images V4 (from Google): 9 million images in 19.7 K categories
- BDD100K (from UC Berkeley): Images from 100,000 driving videos, over 1,100 hours
- YFCC100M (from Yahoo): 99.2 million images

Q: *What are some of the readily available large video datasets I could use?*

Name	Details
YouTube-8M	6.1 million videos, 3,862 classes, 2.6 billion audio-visual features 3.0 labels/video 1.53 terabytes of randomly sampled videos
Something Something (from Twenty Billion Neurons)	221,000 videos in 174 action classes For example, “Pouring water into wine glass but missing so it spills next to it” Humans performing predefined actions with everyday objects
Jester (from Twenty Billion Neurons)	148,000 videos in 27 classes For example, “Zooming in with two fingers” Predefined hand gestures in front of a webcam

Q: *Are those the largest labeled datasets ever assembled in the history of time?*

Nope! Companies like Facebook and Google curate their own private datasets that are much larger than the public ones we can play with:

- Facebook: 3.5 billion Instagram images with noisy labels (first reported in 2018)
- Google – JFT-300M: 300 million images with noisy labels (first reported in 2017)

Sadly, unless you're an employee at one of these companies, you can't really access these datasets. Nice recruiting tactic, we must say.

Q: *How can I get help annotating data?*

There are several companies out there that can assist with labeling different kinds of annotations. A few worth mentioning include SamaSource, Digital Data Divide, and iMerit, which employ people who otherwise have limited opportunities, eventually creating positive socioeconomic change through employment in underprivileged communities.

Q: *Is there a versioning tool for datasets, like Git is for code?*

Qri and Quilt can help version control our datasets, aiding in reproducibility of experiments.

Q: *What if I don't have access to a large dataset for my unique problem?*

Try to develop a synthetic dataset for training! For example, find a realistic 3D model of the object of interest and place it in realistic environments using a 3D framework such as Unity. Adjust the lighting and camera position, zoom, and rotation to take snapshots of this object from many angles, generating an endless supply of training data. Alternatively, companies like AI.Reverie, CVEDIA, Neuromation, Cognata, Mostly.ai, and DataGen Tech provide realistic simulations for training needs. One big benefit of synthesized training data is that the labeling process is built into the synthesization process. After all, you would know what you are creating. This automatic labeling can save a lot of money and effort, compared to manual labeling.

Sadly, unless you're an employee at one of these companies, you can't really access these datasets. Nice recruiting tactic, we must say.

Q: *How can I get help annotating data?*

There are several companies out there that can assist with labeling different kinds of annotations. A few worth mentioning include SamaSource, Digital Data Divide, and iMerit, which employ people who otherwise have limited opportunities, eventually creating positive socioeconomic change through employment in underprivileged communities.

Q: *Is there a versioning tool for datasets, like Git is for code?*

Qri and Quilt can help version control our datasets, aiding in reproducibility of experiments.

Q: *What if I don't have access to a large dataset for my unique problem?*

Try to develop a synthetic dataset for training! For example, find a realistic 3D model of the object of interest and place it in realistic environments using a 3D framework such as Unity. Adjust the lighting and camera position, zoom, and rotation to take snapshots of this object from many angles, generating an endless supply of training data. Alternatively, companies like AI.Reverie, CVEDIA, Neuromation, Cognata, Mostly.ai, and DataGen Tech provide realistic simulations for training needs. One big benefit of synthesized training data is that the labeling process is built into the synthesization process. After all, you would know what you are creating. This automatic labeling can save a lot of money and effort, compared to manual labeling.

Privacy

Q: *How do I develop a more privacy-preserving model without going down the cryptography rabbit hole?*

TensorFlow Encrypted might be the solution you're looking for. It enables development using encrypted data, which is relevant, especially if you are on the cloud. Internally, lots of secure multiparty computation and homomorphic encryptions result in privacy-preserving machine learning.

Q: *Can I keep my model under wraps from prying eyes?*

Well, unless you are on the cloud, weights are visible and can be reverse engineered. Use the Fritz library for protecting your model's IP when deployed on smartphones.

Privacy

Q: *How do I develop a more privacy-preserving model without going down the cryptography rabbit hole?*

TensorFlow Encrypted might be the solution you're looking for. It enables development using encrypted data, which is relevant, especially if you are on the cloud. Internally, lots of secure multiparty computation and homomorphic encryptions result in privacy-preserving machine learning.

Q: *Can I keep my model under wraps from prying eyes?*

Well, unless you are on the cloud, weights are visible and can be reverse engineered. Use the Fritz library for protecting your model's IP when deployed on smartphones.

Education and Exploration

Q: *I want to become an AI expert. Beyond this book, where should I invest my time to learn more?*

There are several resources on the internet to learn deep learning in depth. We highly recommend these video lectures from some of the best teachers, covering a variety of application areas from computer vision to natural language processing.

- Fast.ai (by Jeremy Howard and Rachel Thomas) features a free 14-video lecture series, taking a more learn-by-doing approach in PyTorch. Along with the course comes an ecosystem of tools and an active community that has led to many breakthroughs in the form of research papers and ready-to-use code (like three lines of code to train a state-of-the-art network using the fast.ai library).
- Deeplearning.ai (by Andrew Ng) features a five-course "Deep Learning Specialization." It's free of cost (although you could pay a small fee to get a certificate) and will solidify your theoretical foundation further. Dr. Ng's first Coursera course on machine learning has taught more than two million students, and this series continues the tradition of highly approachable content loved by beginners and experts alike.
- We would be remiss if we didn't encourage you to note O'Reilly's Online Learning platform in this list. Helping more than two million users advance their careers, it contains hundreds of books, videos, live online trainings, and keynotes given by leading thinkers and practitioners at O'Reilly's AI and data conferences.

Q: *Where can I find interesting notebooks to learn from?*

Google Seedbank is a collection of interactive machine learning examples. Built on top of Google Colaboratory, these Jupyter notebooks can be run instantly without any installations. Some interesting examples include:

- Generating audio with GANs

- Action recognition on video
- Generating Shakespeare-esque text
- Audio-style transfer

Q: *Where can I learn about the state of the art for a specific topic?*

Considering how fast the state of the art moves in AI, SOTAWHAT is a handy command-line tool to search research papers for the latest models, datasets, tasks, and more. For example, to look up the latest results on ImageNet, use `sotawhat imagenet` on the command line. Additionally, *paperswithcode.com/sota* also features repositories for papers, their source code, and released models, along with an interactive visual timeline of benchmarks.

Q: *I am reading a paper on Arxiv and I really like it. Do I need to write code from scratch?*

Not at all! The ResearchCode Chrome extension makes it easy to find code when browsing *arxiv.org* or Google Scholar. All it takes is a press of the extension button. You can also look up code without installing the extension on the *ResearchCode.com* website.

Q: *I don't want to write any code, but I still want to interactively experiment with a model using my camera. How can I do that?*

Runway ML is an easy-to-use yet powerful GUI tool that allows you to download models (from the internet or your own) and use the webcam or other input, such as video files, to see the output interactively. This allows further combining and remixing outputs of models to make new creations. And all of this happens with just a few mouse clicks; hence, it's attracting a large artist community!

Q: *8-1If I can test without code, can I train without code, too?*

We discuss this in detail in Chapter 8 (web-based) and Chapter 12 (desktop-based). To keep it short, tools such as Microsoft's CustomVision.ai, Google's Cloud AutoML Vision, Clarifai, Baidu EZDL, and Apple's Create ML provide drag-and-drop training capabilities. Some of these tools take as little as a few seconds to do the training.

One Last Question

Q: *Tell me a great deep learning prank?*

Print and hang poster shown in Figure 7-1 from *keras4kindergartners.com* near the watercooler, and watch people's reactions.

Figure 7-1. Satirical poster on the state of AI from keras4kindergartners.com

Chapter 8. Cloud APIs for Computer Vision: Up and Running in 15 Minutes

Due to repeated incidents of near meltdown at the nearby nuclear power plant, the library of the city of Springfield (we are not allowed to mention the state) decided that it was too risky to store all their valuable archives in physical form. After hearing that the library from their rival city of Shelbyville started digitizing their records, they wanted to get in on the game as well. After all, their collection of articles such as “Old man yells at cloud” and “Local man thinks wrestling is real” and the hundred-year-old iconic photographs of the Gorge and the statue of the city’s founder Jebediah Springfield are irreplaceable. In addition to making their archives resilient to catastrophes, they would make their archives easily searchable and retrievable. And, of course, the residents of Springfield now would be able to access all of this material from the comfort of their living room couches.

The first step in digitizing documents is, of course, scanning. That’s the easy part. Then starts the real challenge—processing and understanding all of this visual imagery. The team in Springfield had a few different options in front of them.

- Perform manual data entry for every single page and every single photograph. Given that the city has more than 200 years of rich history, it would take a really long time, and would be error prone and expensive. It would be quite an ordeal to transcribe all of that material.
- Hire a team of data scientists to build an image understanding system. That would be a much better approach, but there’s just one tiny hitch in the plan. For a library that runs on charitable donations, hiring a team of data scientists would quickly exhaust its budget. A single data scientist might no only be the highest-paid employee at the library, they might also be the highest-earning worker in the entire city of Springfield (barring the wealthy

industrialist Montgomery Burns).

- Get someone who knows enough coding to use the intelligence of ready-to-use vision APIs.

Logically they went with the quick and inexpensive third option. They had a stroke of luck, too. Martin Prince, an industrious fourth grader from Springfield Elementary who happened to know some coding, volunteered to build out the system for them. Although Martin did not know much deep learning (he's just 10 years old, after all), he did know how to do some general coding, including making REST API calls using Python. And that was all he really needed to know. In fact, it took him just under 15 minutes to figure out how to make his first API call.

Martin's *modus operandi* was simple: send a scanned image to the cloud API, get a prediction back, and store it in a database for future retrieval. And obviously, repeat this process for every single record the library owned. He just needed to select the correct tool for the job.

All the big names—Amazon, Google, IBM, Microsoft—provide a similar set of computer-vision APIs that label images, detect and recognize faces and celebrities, identify similar images, read text, and sometimes even discern handwriting. Some of them even provide the ability to train our own classifier without having to write a single line of code. Sounds really convenient!

In the background, these companies are constantly working to improve the state of the art in computer vision. They have spent millions in acquiring and labeling datasets with a granular taxonomy much beyond the ImageNet dataset. We mighi as well make good use of their researchers' blood, sweat, and tears (and electricity bills).

The ease of use, speed of onboarding and development, the variety of functionality, richness of tags, and competitive pricing make cloud-based APIs difficult to ignore. And all of this without the need to hire an expensive data science team. Chapters Chapter 5 and Chapter 6 optimized for accuracy and performance, respectively; this chapter essentially optimizes for human resources.

In this chapter, we explore several cloud-based visual recognition APIs. We

compare them all both quantitatively as well as qualitatively. This should hopefully make it easier to choose the one that best suits your target application. And if they still don’t match your needs, we’ll investigate how to train a custom classifier with just a few clicks.

(In the interest of full disclosure, some of the authors of this book were previously employed at Microsoft, whose offerings are discussed here. We have attempted not to let that bias our results by building reproducible experiments and justifying our methodology.)

t

compare them all both quantitatively as well as qualitatively. This should hopefully make it easier to choose the one that best suits your target application. And if they still don’t match your needs, we’ll investigate how to train a custom classifier with just a few clicks.

(In the interest of full disclosure, some of the authors of this book were previously employed at Microsoft, whose offerings are discussed here. We have attempted not to let that bias our results by building reproducible experiments and justifying our methodology.)

The Landscape of Visual Recognition APIs

Let’s explore some of the different visual recognition APIs out there.

The Landscape of Visual Recognition APIs

Let’s explore some of the different visual recognition APIs out there.

Clarifai

Clarifai (Figure 8-1) was the winner of the 2013 ILSVRC classification task. Started by Matthew Zeiler, a graduate student from New York University, this was one of the first visual recognition API companies out there.

> **NOTE**
>
> Fun fact: While investigating a classifier to detect NSFW (Not Safe For Work) images, it became important to understand and debug what was being learned by the CNN in order to reduce false positives. This led Clarifai to invent a visualization technique to expose which images stimulate feature maps at any layer in the CNN. As they say, necessity is the mother of invention.

What's unique about this API?

It offers multilingual tagging in more than 23 languages, visual similarity search among previously uploaded photographs, face-based multicultural appearance classifier, photograph aesthetic scorer, focus scorer, and embedding vector generation to help us build our own reverse-image search. It also offers recognition in specialized domains including clothing and fashion, travel and hospitality, and weddings. Through its public API, the image tagger supports 11,000 concepts.

Figure 8-1. Sample of Clarifai's results

Microsoft Cognitive Services

With the creation of ResNet-152 in 2015, Microsoft was able to win seven tasks at the ILSVRC, the COCO Image Captioning Challenge as well as the Emotion Recognition in the Wild challenge, ranging from classification and detection (localization) to image descriptions. And most of this research was translated to cloud APIs. Originally starting out as Project Oxford from Microsoft Research in 2015, it was eventually renamed Cognitive Services in 2016. It's a comprehensive set of more than 50 APIs ranging from vision, natural language processing, speech, search, knowledge graph linkage, and more. Historically, many of the same libraries were being run at divisions at Xbox and Bing, but they are now being exposed to developers externally. Some viral applications showcasing creative ways developers use these APIs include *how-old.net* (How Old Do I Look?), Mimicker Alarm (which requires making a particular facial expression in order to defuse the morning alarm), and *CaptionBot.ai*.

What's unique about this API?

As illustrated in Figure 8-2, the API offers image captioning, handwriting understanding, and headwear recognition. Due to many enterprise customers, Cognitive Services does not use customer image data for improving its services.

Figure 8-2. Sample of Microsoft Cognitive Services results

n

Google Cloud Vision

Google provided the winning entry at the 2014 ILSVRC with the help of the 22-layer GoogLeNet, which eventually paved the way for the now-staple Inception architectures. Supplementing the Inception models, in December 2015, Google released a suite of Vision APIs. In the world of deep learning, having large amounts of data is definitely an advantage to improve one's classifier, and Google has a lot of consumer data. For example, with learnings from Google Street View, you should expect relatively good performance in real-world text extraction tasks, like on billboards.

What's unique about this API?

For human faces, it provides the most detailed facial key points (Figure 8-3) including roll, tilt, and pan to accurately localize the facial features. The APIs also return similar images on the web to the given input. A simple way to try out the performance of Google's system without writing code is by uploading photographs to Google Photos and searching through the tags.

Figure 8-3. Sample of Google Cloud Vision's results

Google Cloud Vision

Google provided the winning entry at the 2014 ILSVRC with the help of the 22-layer GoogLeNet, which eventually paved the way for the now-staple Inception architectures. Supplementing the Inception models, in December 2015, Google released a suite of Vision APIs. In the world of deep learning, having large amounts of data is definitely an advantage to improve one's classifier, and Google has a lot of consumer data. For example, with learnings from Google Street View, you should expect relatively good performance in real-world text extraction tasks, like on billboards.

What's unique about this API?

For human faces, it provides the most detailed facial key points (Figure 8-3) including roll, tilt, and pan to accurately localize the facial features. The APIs also return similar images on the web to the given input. A simple way to try out the performance of Google's system without writing code is by uploading photographs to Google Photos and searching through the tags.

Figure 8-3. Sample of Google Cloud Vision's results

Amazon Rekognition

No, that title is not a typo. Amazon Rekognition API (Figure 8-4) is largely based on Orbeus, a Sunnyvale, California-based startup that was acquired by Amazon in late 2015. Founded in 2012, its chief scientist also had winning entries in the ILSVRC 2014 detection challenge. The same APIs were used to power PhotoTime, a famous photo organization app. The API's services are available as part of the AWS offerings. Considering most companies already offer photo analysis APIs, Amazon is doubling down on video recognition offerings to offer differentiation.

What's unique about this API?

License plate recognition, video recognition APIs, and better end-to-end integration examples of Rekognition APIs with AWS offerings like Kinesis Video Streams, Lambda, and others. Also, Amazon's API is the only one that can determine whether the subject's eyes are open or closed.

Figure 8-4. Sample of Amazon Rekognition's results

Amazon Rekognition

No, that title is not a typo. Amazon Rekognition API (Figure 8-4) is largely based on Orbeus, a Sunnyvale, California-based startup that was acquired by Amazon in late 2015. Founded in 2012, its chief scientist also had winning entries in the ILSVRC 2014 detection challenge. The same APIs were used to power PhotoTime, a famous photo organization app. The API's services are available as part of the AWS offerings. Considering most companies already offer photo analysis APIs, Amazon is doubling down on video recognition offerings to offer differentiation.

What's unique about this API?

License plate recognition, video recognition APIs, and better end-to-end integration examples of Rekognition APIs with AWS offerings like Kinesis Video Streams, Lambda, and others. Also, Amazon's API is the only one that can determine whether the subject's eyes are open or closed.

Figure 8-4. Sample of Amazon Rekognition's results

IBM Watson Visual Recognition

Under the Watson brand, IBM's Visual Recognition offering started in early 2015. After purchasing AlchemyAPI, a Denver-based startup, AlchemyVision has been used for powering the Visual Recognition APIs (Figure 8-5). Like others, IBM also offers custom classifier training. Surprisingly, Watson does not offer optical character recognition yet.

Watson sees...

JSON

Classes	Score
favorite	0.69
person	0.72
highchair	0.52
chair	0.52
seat	0.52
furniture	0.52
animal breeding	0.52
caretaker	0.50
dog	0.60
Animal Tending	0.60
light brown color	0.77
reddish orange color	0.56

Type Hierarchy

/person/favorite

/furniture/seat/chair/highchair

/person/caretaker

Did We Wow You? Yes No

JSON

Food	Score
cake mix	0.77
ready-mix	0.77
food mix	0.77
food mixture	0.77
pottage	0.50
soup	0.50

Did We Wow You? Yes No

JSON

Faces	Score
age 18 - 24	0.47
female	0.82

Did We Wow You? Yes No

Figure 8-5. Sample of IBM Watson's Visual Recognition results

Algorithmia

Algorithmia is a marketplace for hosting algorithms as APIs on the cloud. Founded in 2013, this Seattle-based startup has both its own in-house algorithms as well as those created by others (in which case creators earn revenue based on the number of calls). In our experience, this API did tend to have the slowest response time.

What's unique about this API?

Colorization service for black and white photos (Figure 8-6), image stylization, image similarity, and the ability to run these services on-premises, or on any cloud provider.

Figure 8-6. Sample of Algorithmia's style transfer results

With so many offerings, it can be overwhelming to choose a service. There are many reasons why we might choose one over another. Obviously, the biggest factors for most developers would be accuracy and price. Accuracy is the big promise that the deep learning revolution brings, and many applications require

it on a consistent basis. Price of the service might be an additional factor to consider. We might also choose a service provider because our company already has a billing account with it, and it would take additional effort to integrate a different service provider. Speed of the API response might be another factor, especially if the user is waiting on the other end for a response. Because many of these API calls can be abstracted, it’s easy to switch between different providers.

it on a consistent basis. Price of the service might be an additional factor to consider. We might also choose a service provider because our company already has a billing account with it, and it would take additional effort to integrate a different service provider. Speed of the API response might be another factor, especially if the user is waiting on the other end for a response. Because many of these API calls can be abstracted, it’s easy to switch between different providers.

Comparing Visual Recognition APIs

To aid our decision making, let’s compare these APIs head to head. In this section, we examine service offerings, cost, and accuracy of each.

Comparing Visual Recognition APIs

To aid our decision making, let’s compare these APIs head to head. In this section, we examine service offerings, cost, and accuracy of each.

Service Offerings

Table 8-1 lists what services are being offered by each cloud provider.

Table 8-1. Comparison shopping of vision API providers (as of Aug. 2019)

	Algorithmia	Amazon Rekognition	Clarifai	Microsoft Cognitive Services	Google Cloud Vision	IBM Watson Visual Recognition
Image classification	✔	✔	✔	✔	✔	✔
Image detection	✔	✔		✔	✔	
OCR	✔	✔		✔	✔	
Face recognition	✔	✔		✔		
Emotion recognition	✔		✔	✔	✔	
Logo recognition			✔	✔	✔	
Landmark recognition			✔	✔	✔	✔
Celebrity recognition	✔	✔	✔	✔	✔	✔
Multilingual tagging			✔			
Image description				✔		
Handwriting				✔	✔	
Thumbnail generation	✔			✔	✔	
Content moderation	✔	✔	✔	✔	✔	
Custom classification training			✔	✔	✔	✔

Custom detector training				✔	✔	
Mobile custom models			✔	✔	✔	
Free tier	5,000 requests per month	5,000 requests per month	5,000 requests per month	5,000 requests per month	1,000 requests per month	7,500

That’s a mouthful of services already up and running, ready to be used in our application. Because numbers and hard data help make decisions easier, it’s time to analyze these services on two factors: cost and accuracy.

Custom detector training				✔	✔	
Mobile custom models			✔	✔	✔	
Free tier	5,000 requests per month	5,000 requests per month	5,000 requests per month	5,000 requests per month	1,000 requests per month	7,500

That’s a mouthful of services already up and running, ready to be used in our application. Because numbers and hard data help make decisions easier, it’s time to analyze these services on two factors: cost and accuracy.

Cost

Money doesn't grow on trees (yet), so it's important to analyze the economics of using off-the-shelf APIs. Taking a heavy-duty example of querying these APIs a about 1 query per second (QPS) service for one full month (roughly 2.6 million requests per month), Figure 8-7 presents a comparison of the different providers sorted by estimated costs (as of August 2019).

Figure 8-7. A cost comparison of different cloud-based vision APIs

Although for most developers, this is an extreme scenario, this would be a pretty realistic load for large corporations. We will eventually compare these prices against running our own service in the cloud to make sure we get the most bang for the buck fitting our scenario.

That said, many developers might find negligible charges, considering that all of the cloud providers we look at here have a free tier of 5,000 calls per month (except Google Vision, which gives only 1,000 calls per month for free), and then roughly $1 per 1,000 calls.

Cost

Money doesn't grow on trees (yet), so it's important to analyze the economics of using off-the-shelf APIs. Taking a heavy-duty example of querying these APIs at about 1 query per second (QPS) service for one full month (roughly 2.6 million requests per month), Figure 8-7 presents a comparison of the different providers sorted by estimated costs (as of August 2019).

Figure 8-7. A cost comparison of different cloud-based vision APIs

Although for most developers, this is an extreme scenario, this would be a pretty realistic load for large corporations. We will eventually compare these prices against running our own service in the cloud to make sure we get the most bang for the buck fitting our scenario.

That said, many developers might find negligible charges, considering that all of the cloud providers we look at here have a free tier of 5,000 calls per month (except Google Vision, which gives only 1,000 calls per month for free), and then roughly $1 per 1,000 calls.

Accuracy

In a world ruled by marketing departments who claim their organizations to be the market leaders, how do we judge who is actually the best? What we need are common metrics to compare these service providers on some external datasets.

To showcase building a reproducible benchmark, we assess the text extraction quality using the COCO-Text dataset, which is a subset of the MS COCO dataset. This 63,686-image set contains text in daily life settings, like on a banner, street sign, number on a bus, price tag in a grocery store, designer shirt, and more. This real-world imagery makes it a relatively tough set to test against. We use the Word Error Rate (WER) as our benchmarking metric. To keep things simple, we ignore the position of the word and focus only on whether a word is present (i.e., bag of words). To be a match, the entire word must be correct.

In the COCO-Text validation dataset, we pick all images with one or more instances of legible text (full-text sequences without interruptions) and compare text instances of more than one-character length. We then send these images to various cloud vision APIs. Figure 8-8 presents the results.

Figure 8-8. WER for different text extraction APIs as of August 2019

Considering how difficult the dataset is, these results are remarkable. Most state-of-the-art text extraction tools from earlier in the decade would not cross the 10% mark. This shows the power of deep learning. On a subset of manually tested images, we also noticed a year-on-year improvement in the performance of some of these APIs, which is another benefit enjoyed by cloud-based APIs.

As always, all of the code that we used for our experiment is hosted on GitHub

(see *http://PracticalDeepLearning.ai*.

The results of our analysis depend significantly on the dataset we choose as well as our metrics. Depending on our dataset (which is in turn influenced by our use case) as well as our minimum quality metrics, our results can vary. Additionally, service providers are constantly improving their services in the background. As a consequence, these results are not set in stone and improve over time. These results can be replicated on any dataset with the scripts on GitHub.

(see *http://PracticalDeepLearning.ai*.

The results of our analysis depend significantly on the dataset we choose as well as our metrics. Depending on our dataset (which is in turn influenced by our use case) as well as our minimum quality metrics, our results can vary. Additionally, service providers are constantly improving their services in the background. As a consequence, these results are not set in stone and improve over time. These results can be replicated on any dataset with the scripts on GitHub.

Bias

In Chapter 1, we explored how bias can creep into datasets and how it can have real-life consequences for people. The APIs we explore in this chapter are no exception. Joy Buolamwini, a researcher at the MIT Media Lab, discovered that among Microsoft, IBM, and Megvii (also known as Face++), none were able to detect her face and gender accurately. Wondering if she had unique facial features that made her undetectable to these APIs, she (working along with Timnit Gebru) compiled faces of members of legislative branches from six countries with a high representation of women, building the Pilot Parliaments Benchmark (PPB; see Figure 8-9). She chose members from three African countries and three European countries to test for how the APIs performed on different skin tones. If you haven't been living under a rock, you can already see where this is going.

She observed that the APIs performed fairly well overall at accuracies between 85% and 95%. It was only when she started slicing the data across the different categories that she observed there was a massive amount of difference in the accuracies for each. She first observed that there was a significant difference between detection accuracies of men and women. She also observed that breaking down by skin tone, the difference in the detection accuracy was even larger. Then, finally, taking both gender and skin tone into consideration, the differences grew painfully starker between the worse detected group (darker females) and the best detected group (lighter males). For example, in the case of IBM, the detection accuracy of African women was a mere 65.3%, whereas the same API gave a 99.7% accuracy for European men. A whopping 34.4% difference! Considering many of these APIs are used by law enforcement, the consequences of bias seeping in might have life or death consequences.

Figure 8-9. Averaged faces among different gender and skin tone, from Pilot Parliaments Benchmark (PPB

Following are a few insights we learned from this study:

- The algorithm is only as good as the data on which it's trained. And this shows the need for diversity in the training dataset.
- Often the aggregate numbers don't always reveal the true picture. The bias in the dataset is apparent only when slicing it across different subgroups.
- The bias does not belong to any specific company; rather, it's an industry-wide phenomenon.
- These numbers are not set in stone and reflect only the time at which the experiment was performed. As evident from the drastic change in numbers between 2017 (Figure 8-10) and a subsequent study in 2018 (Figure 8-11), these companies are taking bias removal from their datasets quite seriously.
- Researchers putting commercial companies to the test with public benchmarks results in industry-wide improvements (even if for the fear of bad PR, then so be it).

	Overall	Darker Female	Darker Male	Lighter Female	Lighter Male
Microsoft	93.70%	79.20%	94.00%	98.30%	100%
Face++	90.00%	65.50%	99.30%	94%	99.20%
IBM	87.90%	65.30%	88.00%	92.90%	99.70%

Figure 8-10. Face detection comparison across APIs, tested in April and May 2017 on the PPB

	Overall	Darker Female	Darker Male	Lighter Female	Lighter Male
Microsoft	99.52%	98.48%	99.67%	99.66%	100%
Face++	98.40%	95.90%	98.70%	99%	99.50%
IBM	95.59%	83.03%	99.37%	97.63%	99.74%
Kairos	93.40%	77.50%	98.70%	93.60%	100%
Amazon	91.34%	68.63%	98.74%	92.88%	100%

Figure 8-11. Face detection comparison across APIs in August 2018 on the PPB, conducted by Inioluwa Deborah Raji et al.

How about bias in image-tagging APIs? Facebook AI Research pondered over the question “Does Object Recognition Work for Everyone?” in a paper by the same title (Terrance DeVries et al.). The group tested multiple cloud APIs in February 2019 on Dollar Street, a diverse collection of images of household items from 264 different homes across 50 countries (Figure 8-12).

Figure 8-12. Image-tagging API performance on geographically diverse images from the Dollar Street dataset

Here are some of the key learnings from this test:

- Accuracy of object classification APIs was significantly lower in images from regions with lower income levels, as illustrated in Figure 8-13.
- Datasets such as ImageNet, COCO, and OpenImages severely undersample images from Africa, India, China, and Southeast Asia, hence leading to lower performance on images from the non-Western world.
- Most of the datasets were collected starting with keyword searches in English, omitting images that mentioned the same object with phrases in other languages.

Figure 8-13. Average accuracy (and standard deviation) of six cloud APIs versus income of the household where the images were collected

In summary, depending on the scenario for which we want to use these cloud APIs, we should build our own benchmarks and test them periodically to evaluate whether these APIs are appropriate for the use case.

Getting Up and Running with Cloud APIs

Calling these cloud services requires minimal code. At a high level, get an API key, load the image, specify the intent, make a POST request with the proper encoding (e.g., base64 for the image), and receive the results. Most of the cloud providers offer software development kits (SDKs) and sample code showcasing how to call their services. They additionally provide pip-installable Python packages to further simplify calling them. If you're using Amazon Rekognition, we highly recommend using its `pip` package.

Let's reuse our thrilling image to test-run these services.

First, let's try it on Microsoft Cognitive Services. Get an API key and replace it in the following code (the first 5,000 calls are free—more than enough for our experiments):

```
cognitive_services_tagimage('DogAndBaby.jpg')

Results:

{
    "description": {
        "tags": ["person", "indoor", "sitting", "food", "table", "little",
"small", "dog", "child", "looking", "eating", "baby", "young", "front",
"feeding", "holding", "playing", "plate", "boy", "girl", "cake", "bowl",
"woman", "kitchen", "standing", "birthday", "man", "pizza"],
        "captions": [{
            "text": "a little girl sitting at a table with a dog",
            "confidence": 0.84265453815486435
        }]
    },
    "requestId": "1a32c16f-fda2-4adf-99b3-9c4bf9e11a60",
    "metadata": {
        "height": 427,
        "width": 640,
        "format": "Jpeg"
    }
}
```

"A little girl sitting at a table with a dog"—pretty close! There are other options to generate more detailed results, including a probability along with each tag.

TIP

Although the ImageNet dataset is primarily tagged with nouns, many of these services go beyond and return verbs like “eating,” “sitting,” “jumping.” Additionally, they might contain adjectives like “red.” Chances are, these might not be appropriate for our application. We might want to filter out these adjectives and verbs. One option is to check their linguistic type against Princeton’s WordNet. This is available in Python with the Natural Language Processing Toolkit (NLTK). Additionally, we might want to filter out words like “indoor” and “outdoor” (often shown by Clarifai and Cognitive Services).

Now, let’s test the same image using Google Vision APIs. Get an API key from their website and use it in the following code (and rejoice, because the first 1,00(calls are free):

```
google_cloud_tagimage('DogAndBaby.jpg')

Results:

{
 "responses": [
    {
      "labelAnnotations": [
        {
          "mid": "/m/0bt9lr",
          "description": "dog",
          "score": 0.951077,
          "topicality": 0.951077
        },
        {
          "mid": "/m/06z04",
          "description": "skin",
          "score": 0.9230451,
          "topicality": 0.9230451
        },
        {
          "mid": "/m/01z5f",
          "description": "dog like mammal",
          "score": 0.88359463,
          "topicality": 0.88359463
        },
        {
          "mid": "/m/01f5gx",
          "description": "eating",
          "score": 0.7258142,
          "topicality": 0.7258142
        }
```

```
  # other objects
        ]
      }
   ]
  }
```

Wasn’t that a little too easy? These APIs help us get to state-of-the-art results without needing a Ph.D.—in just 15 minutes!

TIP

Even though these services return tags and image captions with probabilities, it’s up to the developer to determine a threshold. Usually, 60% and 40% are good thresholds for image tags and image captions, respectively.

It’s also important to communicate the probability to the end-user from a UX standpoint. For example, if the result confidence is >80%, we might say prefix the tags with “This image *contains*....” For <80%, we might want to change that prefix to “This image *may contain*…” to reflect the lower confidence in the result.

```
  # other objects
        ]
      }
   ]
  }
```

Wasn’t that a little too easy? These APIs help us get to state-of-the-art results without needing a Ph.D.—in just 15 minutes!

TIP

Even though these services return tags and image captions with probabilities, it’s up to the developer to determine a threshold. Usually, 60% and 40% are good thresholds for image tags and image captions, respectively.

It’s also important to communicate the probability to the end-user from a UX standpoint. For example, if the result confidence is >80%, we might say prefix the tags with “This image *contains*....” For <80%, we might want to change that prefix to “This image *may contain*…” to reflect the lower confidence in the result.

Training Our Own Custom Classifier

Chances are these services were not quite sufficient to meet the requirements of our use case. Suppose that the photograph we sent to one of these services responded with the tag “dog.” We might be more interested in identifying the breed of the dog. Of course, we can follow Chapter 3 to train our own classifier in Keras. But wouldn’t it be more awesome if we didn’t need to write a single line of code? Help is on the way.

A few of these cloud providers give us the ability to train our own custom classifier by merely using a drag-and-drop interface. The pretty user interfaces provide no indication that under the hood they are using transfer learning. As a result, Cognitive Services Custom Vision, Google AutoML, Clarifai, and IBM Watson all provide us the option for custom training. Additionally, some of them even allow building custom detectors, which can identify the location of objects with a bounding box. The key process in all of them being the following:

1. Upload images
2. Label them
3. Train a model
4. Evaluate the model
5. Publish the model as a REST API
6. Bonus: Download a mobile-friendly model for inference on smartphones and edge devices

Let’s see a step-by-step example of Microsoft’s Custom Vision.

1. *Create a project* (Figure 8-14): Choose a domain that best describes our use case. For most purposes, “General” would be optimal. For more specialized scenarios, we might want to choose a relevant domain.

Figure 8-14. Creating a new project in Custom Vision

As an example, if we have an ecommerce website with photos of products against a pure white background, we might want to select the “Retail” domain. If we intend to run this model on a mobile phone eventually, we

should choose the “Compact” version of the model, instead; it is smaller in size with only a slight loss in accuracy.

2. *Upload* (Figure 8-15): For each category, upload images and tag them. It’s important to upload at least 30 photographs per category. For our test, we uploaded more than 30 images of Maltese dogs and tagged them appropriately.

Figure 8-15. Uploading images on CustomVision.ai

3. *Train* (Figure 8-16): Click the Train button, and then in about three minutes we have a spanking new classifier ready.

Figure 8-16. The Train button in the upper-right corner of the CustomVision.ai page

4. *Analyze the model's performance*: Check the precision and recall of the model. By default, the system sets the threshold at 90% confidence and gives the precision and recall metrics at that value. For higher precision, increase the confidence threshold. This would come at the expense of reduced recall. Figure 8-17 shows example output.

5. *Ready to go*: We now have a production-ready API endpoint that we can call from any application.

To highlight the effect of the amount of data on model quality, let's train a dog breed classifier. We can use the Stanford Dogs dataset, a collection of more than 100 dog categories. For simplicity, we randomly chose 10 breeds, which have more than 200 images available. With 10 classes, a random classifier would have one-tenth, or 10%, the chance of correctly identifying an image. We should easily be able to beat this number. Table 8-2 shows the effect of training on datasets with different volumes.

Table 8-2. Effect of number of training images on precision and recall

	30 training images/class	200 training images/class
Precision	91.2%	93.5%
Recall	85.3%	89.6%

,

Figure 8-17. Relative precision and recall for our sample training set with 200 images per class

Because we haven't uploaded a test set, the performance figures reported here are on the full dataset using the common *k*-fold cross-validation technique. This means the data was randomly divided into *k* parts, then (*k* – 1) parts were used for training, and the remaining part was used for testing. This was performed a few times, each time with a randomized subset of images, and the averaged results are reported here.

It is incredible that even with 30 images per class, the classifier's precision is greater than 90%, as depicted in Figure 8-18. And, surprisingly, this took slightly less than 30 seconds to train.

Not only this, we can dig down and investigate the performance on each class. Classes with high precision might visibly be more distinct, whereas those with low precision might look similar to another class.

Performance Per Tag

Tag	Precision	Recall
afghan_hound	87.5%	92.0%
airedale	96.0%	92.5%
basenji	97.4%	93.0%
bernese_mountain_dog	91.3%	91.0%
entlebucher	97.2%	87.5%
great_pyrenees	87.7%	85.0%
irish_wolfhound	87.8%	85.0%
leonberg	98.9%	87.0%
maltese_dogs	96.4%	91.5%
pomeranian	97.4%	91.5%

Figure 8-18. Some of the possible tags returned by the API

This short and convenient approach is not without its downsides, as you will see in the following section. In that section, we also discuss mitigation strategies to help take advantage of this rather useful tool.

Performance Per Tag

Tag	Precision	Recall
afghan_hound	87.5%	92.0%
airedale	96.0%	92.5%
basenji	97.4%	93.0%
bernese_mountain_dog	91.3%	91.0%
entlebucher	97.2%	87.5%
great_pyrenees	87.7%	85.0%
irish_wolfhound	87.8%	85.0%
leonberg	98.9%	87.0%
maltese_dogs	96.4%	91.5%
pomeranian	97.4%	91.5%

Figure 8-18. Some of the possible tags returned by the API

This short and convenient approach is not without its downsides, as you will see in the following section. In that section, we also discuss mitigation strategies to help take advantage of this rather useful tool.

Top Reasons Why Our Classifier Does Not Work Satisfactorily

There are a number of reasons why a classifier would not perform well. The following are some of them:

Not enough data

If we find that the accuracy is not quite sufficient for our needs, we might need to train the system with more data. Of course, 30 images per class just gets us started. But for a production-quality application, more images are better. 200 images per class are usually recommended.

Nonrepresentative training data

Often, the images on the internet are far too clean, set up in studio lighting with clean backgrounds, and close to the center of the frame. Images that our application might see on a daily basis might not be represented quite so well. It's really important to train our classifier with real-world images for the best performance.

Unrelated domain

Under the hood, Custom Vision is running transfer learning. This makes it really important to choose the correct domain when creating the project. As an example, if we are trying to classify X-ray images, transfer learning from an ImageNet-based model might not yield as accurate a result. For cases like that, training our own classifier manually in Keras would work best, as demonstrated in Chapter 3 (though this will probably take more than three minutes).

Using it for regression

In machine learning, there are two common categories of problems: classification and regression. Classification is predicting one or more classes for input. Regression, on the other hand, is predicting a numerical value given an input; for example, predicting house prices. Custom Vision is primarily a classification system. Using it to count objects by tagging the number of objects is the wrong approach, and will lead to unsatisfactory results.

Counting objects is a type of regression problem. We can do it by localizing each instance of the object in an image (aka object detection) and counting their occurrences. Another example of a regression problem is predicting the age of a person based on their profile photo. We tackle both problems in later chapters.

Classes are too similar

If our classes look too similar and rely heavily on smaller-level details for distinction, the model might not perform as well. For example, a five-dollar note and a 20-dollar note have very similar high-level features. It's at the lower-level details that show they are really distinct. As another example, it might be easy to distinguish between a Chihuahua and a Siberian Husky, but it's more difficult to distinguish between an Alaskan Malamute and a Siberian Husky. A fully retrained CNN, as demonstrated in Chapter 3, should perform better than this Custom Vision-based system.

TIP

A great feature of Custom Vision is that if the model is unsure of any image that it encounters via its API endpoint, the web UI will show those images for a manual review. We can review and manually tag new images on a periodic basis and continuously improve the quality of the model. These images tend to improve the classifier the most for two reasons: first, they represent real-world usage. Second, and more importantly, they have more impact on the model in comparison to images that the model can already easily classify. This is known as semisupervised learning.

In this section, we discussed a few different ways in which we can improve our model's accuracy. In the real world, that is not the end-all-be-all of a user's experience. How quickly we are able to respond to a request also matters a lot. In the following section, we cover a couple of different ways we can improve performance without sacrificing quality.

Counting objects is a type of regression problem. We can do it by localizing each instance of the object in an image (aka object detection) and counting their occurrences. Another example of a regression problem is predicting the age of a person based on their profile photo. We tackle both problems in later chapters.

Classes are too similar

If our classes look too similar and rely heavily on smaller-level details for distinction, the model might not perform as well. For example, a five-dollar note and a 20-dollar note have very similar high-level features. It's at the lower-level details that show they are really distinct. As another example, it might be easy to distinguish between a Chihuahua and a Siberian Husky, but it's more difficult to distinguish between an Alaskan Malamute and a Siberian Husky. A fully retrained CNN, as demonstrated in Chapter 3, should perform better than this Custom Vision-based system.

TIP

A great feature of Custom Vision is that if the model is unsure of any image that it encounters via its API endpoint, the web UI will show those images for a manual review. We can review and manually tag new images on a periodic basis and continuously improve the quality of the model. These images tend to improve the classifier the most for two reasons: first, they represent real-world usage. Second, and more importantly, they have more impact on the model in comparison to images that the model can already easily classify. This is known as semisupervised learning.

In this section, we discussed a few different ways in which we can improve our model's accuracy. In the real world, that is not the end-all-be-all of a user's experience. How quickly we are able to respond to a request also matters a lot. In the following section, we cover a couple of different ways we can improve performance without sacrificing quality.

Comparing Custom Classification APIs

As you might have noticed throughout the book, we are pretty dogmatic about being data driven. If we are going to spend good money on a service, we better get the best bang for our buck. Time to put the hype to the test.

For a good number of classification problems, these custom cloud-based classifiers perform pretty well. To truly test their limits, we need something more challenging. We need to unleash the toughest doggone dataset, train this animal, and fetch some insightful results—using the Stanford Dogs dataset.

Using the entire dataset might make it too easy for these classifiers (after all, ImageNet already has so many dog breeds), so we took it up a notch. Instead, we trained our own Keras classifier on the entire dataset and built a mini-dataset out of the top 34 worst-performing classes (each containing at least 140 images). The reason these classes performed poorly was because they often became confused with other similar-looking dog breeds. To perform better, they require a fine-grained understanding of the features. We divide the images into 100 randomly chosen images per class in the training dataset and 40 randomly chosen images per class in the test dataset. To avoid any class imbalances, which can have an impact on predictions, we chose the same number of train and test images for each class.

Lastly, we selected a minimum confidence threshold of 0.5 as it appeared to strike a good balance between precision and recall across all services. At a high confidence threshold such as 0.99, a classifier might be very accurate, but there might be only a handful of images with predictions; in other words, really low recall. On the other hand, a really low threshold of 0.01 would result in predictions for nearly all images. However, we should not rely on many of these results. After all, the classifier is not confident.

Instead of reporting the precision and recall, we report the *F1 score* (also known as *F-measure*), which is a hybrid score that combines both of those values:

$$F1\,score = \frac{2 \times precision \times recall}{precision + recall}$$

Additionally, we report the time it took to train, as shown in Figure 8-19. Beyond just the cloud, we also trained using Apple's Create ML tool on a MacBook Pro with and without data augmentations (rotate, crop, and flip).

Google and Microsoft provide the ability to customize the duration of training. Google Auto ML allows us to customize between 1 and 24 hours. Microsoft provides a free "Fast Training" option and a paid "Advanced Training" option (similar to Google's offering) with which we can select the duration to be anywhere between 1 and 24 hours.

Figure 8-19. A chart showing the F1 score for custom classifier services, as of August 2019 (higher is better)

Following are some interesting takeaways from this experiment:

- Clarifai and Microsoft offered near-instant training for the 3,400 training images.
- Compared to "Fast Training," Microsoft's "Advanced Training" performed slightly better (roughly a 1-point increase) for the extra one hour of training. Because "Fast Training" took less than 15 seconds to train, we can infer that its base featurizer was already good at extracting fine-grained features.
- Surprisingly, Apple's Create ML actually performed worse after adding in the augmentations, despite taking more than two extra hours to train, most of which was spent creating the augmentations. This was done on a top-of-the-line MacBook Pro and showed 100% GPU utilization in Activity Monitor.

Additionally, to test the featurizer's strength, we varied the amount of training data supplied to the service (Figure 8-20). Because Microsoft took less than 15 seconds to train, it was easy (and cheap!) for us to perform the experiment there. We varied between 30 and 100 images per class for training while keeping the same 40 images per class for testing.

Figure 8-20. Effect of varying size of training data per class on test F1 score (higher is better)

Even though Microsoft recommends using at least 50 images per class, going under that limit did not affect performance significantly. The fact that the F1 score did not vary as much as one would expect shows the value of transfer learning (enabling less data to build classifiers) and having a good featurizer capable of fine-grained classification.

It bears repeating that this experiment was intentionally made difficult to stress-test these classifiers. On average, they would have performed much better on the entire Stanford Dogs dataset.

Additionally, to test the featurizer's strength, we varied the amount of training data supplied to the service (Figure 8-20). Because Microsoft took less than 15 seconds to train, it was easy (and cheap!) for us to perform the experiment there. We varied between 30 and 100 images per class for training while keeping the same 40 images per class for testing.

Figure 8-20. Effect of varying size of training data per class on test F1 score (higher is better)

Even though Microsoft recommends using at least 50 images per class, going under that limit did not affect performance significantly. The fact that the F1 score did not vary as much as one would expect shows the value of transfer learning (enabling less data to build classifiers) and having a good featurizer capable of fine-grained classification.

It bears repeating that this experiment was intentionally made difficult to stress-test these classifiers. On average, they would have performed much better on the entire Stanford Dogs dataset.

Performance Tuning for Cloud APIs

A photograph taken by a modern cell phone can have a resolution as high as 4000 x 3000 pixels and be upward of 4 MB in size. Depending on the network quality, it can take a few seconds to upload such an image to the service. With each additional second, it can become more and more frustrating for our users. Could we make this faster?

There are two ways to reduce the size of the image:

Resizing

> Most CNNs take an input image with a size of 224 x 224 or 448 x 448 pixels Much of a cell phone photo's resolution would be unnecessary for a CNN. It would make sense to downsize the image prior to sending it over the network, instead of sending a large image over the network and then downsizing it on the server.

Compression

> Most image libraries perform *lossy* compression while saving a file. Even a little bit of compression can go a long way in reducing the size of the image while minimally affecting the quality of the image itself. Compression does introduce noise, but CNNs are usually robust enough to deal with some of it.

Performance Tuning for Cloud APIs

A photograph taken by a modern cell phone can have a resolution as high as 4000 x 3000 pixels and be upward of 4 MB in size. Depending on the network quality, it can take a few seconds to upload such an image to the service. With each additional second, it can become more and more frustrating for our users. Could we make this faster?

There are two ways to reduce the size of the image:

Resizing

> Most CNNs take an input image with a size of 224 x 224 or 448 x 448 pixels. Much of a cell phone photo's resolution would be unnecessary for a CNN. It would make sense to downsize the image prior to sending it over the network, instead of sending a large image over the network and then downsizing it on the server.

Compression

> Most image libraries perform *lossy* compression while saving a file. Even a little bit of compression can go a long way in reducing the size of the image while minimally affecting the quality of the image itself. Compression does introduce noise, but CNNs are usually robust enough to deal with some of it.

Effect of Resizing on Image Labeling APIs

We performed an experiment in which we took more than a hundred diverse unmodified images taken from an iPhone at the default resolution (4032 x3024) and sent them to the Google Cloud Vision API to get labels for each of those images. We then downsized each of the original images in 5% increments (5%, 10%, 15%…95%) and collected the API results for those smaller images, too. We then calculated the agreement rate for each image using the following formula:

$$\%\,agreement\,rate = \frac{number\,of\,labels\,in\,the\,baseline\,image\,also\,present\,in\,test\,ima}{number\,of\,labels\,in\,the\,baseline\,image} \times 100$$

Figure 8-21 shows the results of this experiment. In the figure, the solid line shows the reduction in file size, and the dotted line represents the agreement rate. Our main conclusion from the experiment was that a 60% reduction in resolution led to a 95% reduction in file size, with little change in accuracy compared to the original images.

Figure 8-21. Effect of resizing an image on agreement rate and file size reduction relative to the original image

$\underline{a}$

Figure 8-21. Effect of resizing an image on agreement rate and file size reduction relative to the original image

Effect of Compression on Image Labeling APIs

We repeated the same experiment, but instead of changing the resolution, we changed the compression factor for each image incrementally. In Figure 8-22, the solid line shows the reduction in file size and the dotted line represents the agreement rate. The main takeaway here is that a 60% compression score (or 40% quality) leads to an 85% reduction in file size, with little change in accuracy compared to the original image.

Figure 8-22. Effect of compressing an image on agreement rate and file size reduction relative to the original image

Effect of Compression on Image Labeling APIs

We repeated the same experiment, but instead of changing the resolution, we changed the compression factor for each image incrementally. In Figure 8-22, the solid line shows the reduction in file size and the dotted line represents the agreement rate. The main takeaway here is that a 60% compression score (or 40% quality) leads to an 85% reduction in file size, with little change in accuracy compared to the original image.

Figure 8-22. Effect of compressing an image on agreement rate and file size reduction relative to the original image

Effect of Compression on OCR APIs

We took a document containing 300-plus words at the default resolution of an iPhone (4032 x3024), and sent it to the Microsoft Cognitive Services API to test text recognition. We then compressed it at 5% increments and then sent each image and compressed it. We sent these images to the same API and compared their results against the baseline to calculate the percentage WER. We observed that even setting the compression factor to 95% (i.e., 5% quality of the original image) had no effect on the quality of results.

Effect of Compression on OCR APIs

We took a document containing 300-plus words at the default resolution of an iPhone (4032 x3024), and sent it to the Microsoft Cognitive Services API to test text recognition. We then compressed it at 5% increments and then sent each image and compressed it. We sent these images to the same API and compared their results against the baseline to calculate the percentage WER. We observed that even setting the compression factor to 95% (i.e., 5% quality of the original image) had no effect on the quality of results.

Effect of Resizing on OCR APIs

We repeated the previous experiment, but this time by resizing each image instead of compressing. After a certain point, the WER jumped from none to almost 100%, with nearly all words being misclassified. Retesting this with another document having each word at a different font size showed that all words under a particular font size were getting misclassified. To effectively recognize text, OCR engines need the text to be bigger than a minimum height (a good rule of thumb is larger than 20 pixels). Hence the higher the resolution, the higher the accuracy.

What have we learned?

- For text recognition, compress images heavily, but do not resize.
- For image labeling, a combination of moderate resizing (say, 50%) and moderate compression (say, 30%) should lead to heavy file size reductions (and quicker API calls) without any difference in quality of API results.
- Depending on your application, you might be working with already resized and compressed images. Every processing step can introduce a slight difference in the results of these APIs, so aim to minimize them.

TIP

After receiving an image, cloud APIs internally resize it to fit their own implementation. For us, this means two levels of resizing: we first resize an image to reduce the size, then send it to the cloud API, which further resizes the image. Downsizing images introduces distortion, which is more evident at lower resolutions. We can minimize the effect of distortion by resizing from a higher resolution, which is bigger by a few multiples. For example, resizing 3024x3024 (original) → 302x302 (being sent to cloud) → 224x224 (internally resized by APIs) would introduce much more distortion in the final image compared to 3024x3024 → 896x896 → 224x224. Hence, it's best to find a happy intermediate size before sending the images. Additionally, specifying advanced interpolation options like `BICUBIC` and `LANCZOS` will lead to more accurate representation of the original image in the smaller version.

ə

Case Studies

Some people say that the best things in life don’t come easy. We believe this chapter proves otherwise. In the following section, we take a look at how some tech industry titans use cloud APIs for AI to drive some very compelling scenarios.

Case Studies

Some people say that the best things in life don’t come easy. We believe this chapter proves otherwise. In the following section, we take a look at how some tech industry titans use cloud APIs for AI to drive some very compelling scenarios.

The New York Times

It might seem like the scenario painted at the beginning of the chapter was taken out of a cartoon, but it was, in fact, pretty close to the case of the *New York Times* (NYT). With more than 160 years of illustrious history, NYT has a treasure trove of photographs in its archives. It stored many of these artifacts in the basement of its building three stories below the ground level, aptly called the "morgue." The value of this collection is priceless. In 2015, due to a plumbing leak, parts of the basement were damaged including some of these archived records. Thankfully the damage was minimal. However, this prompted NYT to consider digitally archiving them to protect against another catastrophe.

The photographs were scanned and stored in high quality. However, the photographs themselves did not have any identifying information. What many of them did have were handwritten or printed notes on the backside giving context for the photographs. NYT used the Google Vision API to scan this text and tag the respective images with that information. Additionally, this pipeline provided opportunities to extract more metadata from the photographs, including landmark recognition, celebrity recognition, and so on. These newly added tags powered its search feature so that anyone within the company and outside could explore the gallery and search using keywords, dates, and so on without having to visit the morgue, three stories down.

The New York Times

It might seem like the scenario painted at the beginning of the chapter was taken out of a cartoon, but it was, in fact, pretty close to the case of the *New York Times* (NYT). With more than 160 years of illustrious history, NYT has a treasure trove of photographs in its archives. It stored many of these artifacts in the basement of its building three stories below the ground level, aptly called the "morgue." The value of this collection is priceless. In 2015, due to a plumbing leak, parts of the basement were damaged including some of these archived records. Thankfully the damage was minimal. However, this prompted NYT to consider digitally archiving them to protect against another catastrophe.

The photographs were scanned and stored in high quality. However, the photographs themselves did not have any identifying information. What many of them did have were handwritten or printed notes on the backside giving context for the photographs. NYT used the Google Vision API to scan this text and tag the respective images with that information. Additionally, this pipeline provided opportunities to extract more metadata from the photographs, including landmark recognition, celebrity recognition, and so on. These newly added tags powered its search feature so that anyone within the company and outside could explore the gallery and search using keywords, dates, and so on without having to visit the morgue, three stories down.

Uber

Uber uses Microsoft Cognitive Services to identify each of its seven million-plus drivers in a couple of milliseconds. Imagine the sheer scale at which Uber must operate its new feature called “Real-Time ID Check.” This feature verifies that the current driver is indeed the registered driver by prompting them to take a selfie either randomly or every time they are assigned to a new rider. This selfie is compared to the driver’s photo on file, and only if the face models are a match is the driver allowed to continue. This security feature is helpful for building accountability by ensuring the security of the passengers and by ensuring that the driver’s account is not compromised. This safety feature is able to detect changes in the selfie, including a hat, beard, sunglasses, and more, and then prompts the driver to take a selfie without the hat or sunglasses.

Figure 8-23. The Uber Drivers app prompts the driver to take a selfie to verify the identity of the driver (image source)

Uber

Uber uses Microsoft Cognitive Services to identify each of its seven million-plus drivers in a couple of milliseconds. Imagine the sheer scale at which Uber must operate its new feature called “Real-Time ID Check.” This feature verifies that the current driver is indeed the registered driver by prompting them to take a selfie either randomly or every time they are assigned to a new rider. This selfie is compared to the driver’s photo on file, and only if the face models are a match is the driver allowed to continue. This security feature is helpful for building accountability by ensuring the security of the passengers and by ensuring that the driver’s account is not compromised. This safety feature is able to detect changes in the selfie, including a hat, beard, sunglasses, and more, and then prompts the driver to take a selfie without the hat or sunglasses.

Figure 8-23. The Uber Drivers app prompts the driver to take a selfie to verify the identity of the driver (image source)

Giphy

Back in 1976, when Dr. Richard Dawkins coined the term “meme,” little did he know it would take on a life of its own four decades later. Instead of giving a simple textual reply, we live in a generation where most chat applications suggest an appropriate animated GIF matching the context. Several applications provide a search specific to memes and GIFs, such as Tenor, Facebook messenger, Swype, and Swiftkey. Most of them search through Giphy (Figure 8-24), the world’s largest search engine for animated memes commonly in the GIF format.

Figure 8-24. Giphy extracts text from animations as metadata for searching

GIFs often have text overlaid (like the dialogue being spoken) and sometimes we want to look for a GIF with a particular dialogue straight from a movie or TV show. For example, the image in Figure 8-24 from the 2010 *Futurama* episode in which the “eyePhone” (sic) was released is often used to express excitement toward a product or an idea. Having an understanding of the contents makes the GIFs more searchable. To make this happen, Giphy uses Google’s Vision API to extract the recognize text and objects—aiding the search for the perfect GIF.

It’s obvious that tagging GIFs is a difficult task because a person must sift through millions of these animations and manually annotate them frame by frame. In 2017, Giphy figured out two solutions to automate this process. The first approach was to detect text from within the image. The second approach was to generate tags based on the objects in the image to supplement the metadata for their search engine. This metadata is stored and searched using ElasticSearch to make a scalable search engine.

For text detection, the company used the OCR services from the Google Vision API on the first frame from the GIFs to confirm whether the GIF actually contained text. If the API replied in the affirmative, Giphy would send the next frames, receive their OCR-detected texts, and figure out the differences in the text; for instance, whether the text was static (remaining the same throughout the duration of the gif) or dynamic (different text in different frames). For generating the class labels corresponding to objects in the image, engineers had two options: label detection or web entities, both of which are available on Google Vision API. Label detection, as the name suggests, provides the actual class name of the object. Web entities provides an entity ID (which can be referenceable in the Google Knowledge Graph), which is the unique web URL for identical and similar images seen elsewhere on the net. Using these additional annotations gave the new system an increase in the click-through-rate (CTR) by 32%. Medium-to-long-tail searches (i.e., not-so-frequent searches) benefitted the most, becoming richer with relevant content as the extracted metadata surfaced previously unannotated GIFs that would have otherwise been hidden. Additionally, this metadata and click-through behavior of users provides data to make a similarity and deduplication feature.

It’s obvious that tagging GIFs is a difficult task because a person must sift through millions of these animations and manually annotate them frame by frame. In 2017, Giphy figured out two solutions to automate this process. The first approach was to detect text from within the image. The second approach was to generate tags based on the objects in the image to supplement the metadata for their search engine. This metadata is stored and searched using ElasticSearch to make a scalable search engine.

For text detection, the company used the OCR services from the Google Vision API on the first frame from the GIFs to confirm whether the GIF actually contained text. If the API replied in the affirmative, Giphy would send the next frames, receive their OCR-detected texts, and figure out the differences in the text; for instance, whether the text was static (remaining the same throughout the duration of the gif) or dynamic (different text in different frames). For generating the class labels corresponding to objects in the image, engineers had two options: label detection or web entities, both of which are available on Google Vision API. Label detection, as the name suggests, provides the actual class name of the object. Web entities provides an entity ID (which can be referenceable in the Google Knowledge Graph), which is the unique web URL for identical and similar images seen elsewhere on the net. Using these additional annotations gave the new system an increase in the click-through-rate (CTR) by 32%. Medium-to-long-tail searches (i.e., not-so-frequent searches) benefitted the most, becoming richer with relevant content as the extracted metadata surfaced previously unannotated GIFs that would have otherwise been hidden. Additionally, this metadata and click-through behavior of users provides data to make a similarity and deduplication feature.

OmniEarth

OmniEarth is a Virginia-based company that specializes in collecting, analyzing, and combining satellite and aerial imagery with other datasets to track water usage across the country, scalably, and at high speeds. The company is able to scan the entire United States at a total of 144 million parcels of land within hours. Internally, it uses the IBM Watson Visual Recognition API to classify images of land parcels for valuable information like how green it is. Combining this classification with other data points such as temperature and rainfall, OmniEarth can predict how much water was used to irrigate the field.

For house properties, it infers data points from the image such as the presence of pools, trees, or irrigable landscaping to predict the amount of water usage. The company even predicted where water is being wasted due to malpractices like overwatering or leaks. OmniEarth helped the state of California understand water consumption by analyzing more than 150,000 parcels of land, and then devised an effective strategy to curb water waste.

OmniEarth

OmniEarth is a Virginia-based company that specializes in collecting, analyzing, and combining satellite and aerial imagery with other datasets to track water usage across the country, scalably, and at high speeds. The company is able to scan the entire United States at a total of 144 million parcels of land within hours. Internally, it uses the IBM Watson Visual Recognition API to classify images of land parcels for valuable information like how green it is. Combining this classification with other data points such as temperature and rainfall, OmniEarth can predict how much water was used to irrigate the field.

For house properties, it infers data points from the image such as the presence of pools, trees, or irrigable landscaping to predict the amount of water usage. The company even predicted where water is being wasted due to malpractices like overwatering or leaks. OmniEarth helped the state of California understand water consumption by analyzing more than 150,000 parcels of land, and then devised an effective strategy to curb water waste.

Photobucket

Photobucket is a popular online image- and video-hosting community where more than two million images are uploaded every day. Using Clarifai's NSFW models, Photobucket automatically flags unwanted or offensive user-generated content and sends it for further review to its human moderation team. Previously the company's human moderation team was able to monitor only about 1% of the incoming content. About 70% of the flagged images turned out to be unacceptable content. Compared to previous manual efforts, Photobucket identified 700 times more unwanted content, thus cleaning the website and creating a better UX. This automation also helped discover two child pornography accounts, which were reported to the FBI.

Photobucket

Photobucket is a popular online image- and video-hosting community where more than two million images are uploaded every day. Using Clarifai's NSFW models, Photobucket automatically flags unwanted or offensive user-generated content and sends it for further review to its human moderation team. Previously, the company's human moderation team was able to monitor only about 1% of the incoming content. About 70% of the flagged images turned out to be unacceptable content. Compared to previous manual efforts, Photobucket identified 700 times more unwanted content, thus cleaning the website and creating a better UX. This automation also helped discover two child pornography accounts, which were reported to the FBI.

Staples

Ecommerce stores like Staples often rely on organic search engine traffic to drive sales. One of the methods to appear high in search engine rankings is to pu descriptive image tags in the ALT text field for the image. Staples Europe, whicl serves 12 different languages, found tagging product images and translating keywords to be an expensive proposition, which is traditionally outsourced to human agencies. Fortunately, Clarifai provides tags in 20 languages at a much cheaper rate, saving Staples costs into five figures. Using these relevant keywords led to an increase in traffic and eventually increased sales through its ecommerce store due to a surge of visitors to the product pages.

Staples

Ecommerce stores like Staples often rely on organic search engine traffic to drive sales. One of the methods to appear high in search engine rankings is to put descriptive image tags in the ALT text field for the image. Staples Europe, which serves 12 different languages, found tagging product images and translating keywords to be an expensive proposition, which is traditionally outsourced to human agencies. Fortunately, Clarifai provides tags in 20 languages at a much cheaper rate, saving Staples costs into five figures. Using these relevant keywords led to an increase in traffic and eventually increased sales through its ecommerce store due to a surge of visitors to the product pages.

InDro Robotics

This Canadian drone company uses Microsoft Cognitive Services to power search and rescue operations, not only during natural disasters but also to proactively detect emergencies. The company utilizes Custom Vision to train models specifically for identifying objects such as boats and life vests in water (Figure 8-25) and use this information to notify control stations. These drones are able to scan much larger ocean spans on their own, as compared to lifeguards. This automation alerts the lifeguard of emergencies, thus improving the speed of discovery and saving lives in the process.

Australia has begun using drones from other companies coupled with inflatable pods to be able to react until help reaches. Soon after deployment, these pods saved two teenagers stranded in the ocean, as demonstrated in Figure 8-26. Australia is also utilizing drones to detect sharks so that beaches can be vacated. It's easy to foresee the tremendous value these automated, custom training services can bring.

Figure 8-25. Detections made by InDro Robotics

Figure 8-26. Drone identifies two stranded swimmers and releases an inflatable pod that they cling onto (image source)

Figure 8-26. Drone identifies two stranded swimmers and releases an inflatable pod that they cling onto (image source)

Summary

In this chapter, we explored various cloud APIs for computer vision, first qualitatively comparing the breadth of services offered and then quantitatively comparing their accuracy and price. We also looked at potential sources of bias that might appear in the results. We saw that with just a short code snippet, we can get started using these APIs in less than 15 minutes. Because one model doesn’t fit all, we trained a custom classifier using a drag-and-drop interface, and tested multiple companies against one another. Finally, we discussed compression and resizing recommendations to speed up image transmission and how they affect different tasks. To top it all off, we examined how companies across industries use these cloud APIs for building real-world applications. Congratulations on making it this far! In the next chapter, we will see how to deploy our own inference server for custom scenarios.

Summary

In this chapter, we explored various cloud APIs for computer vision, first qualitatively comparing the breadth of services offered and then quantitatively comparing their accuracy and price. We also looked at potential sources of bias that might appear in the results. We saw that with just a short code snippet, we can get started using these APIs in less than 15 minutes. Because one model doesn't fit all, we trained a custom classifier using a drag-and-drop interface, and tested multiple companies against one another. Finally, we discussed compression and resizing recommendations to speed up image transmission and how they affect different tasks. To top it all off, we examined how companies across industries use these cloud APIs for building real-world applications. Congratulations on making it this far! In the next chapter, we will see how to deploy our own inference server for custom scenarios.

Chapter 9. Scalable Inference Serving on Cloud with TensorFlow Serving and KubeFlow

Imagine this: you just built a top-notch classifier. Your goal, as the Silicon Valley motto goes, is to "*make the world a better place*," which you're going to do... with a spectacular Dog/Cat classifier. You have a solid business plan and you cannot wait to pitch your magical classifier to that venture capital firm next week. You know that the investors will question you about your cloud strategy, and you need to show a solid demo before they even consider giving you the money. How would you do this? Creating a model is half the battle, serving it is the next challenge, often the bigger one. In fact, for a long time it was common for training a model to only take a few weeks, but trying to serve it to a larger group of people was a months-long battle, often involving backend engineers and DevOps teams.

In this chapter, we answer a few questions that tend to come up in the context of hosting and serving custom-built models.

- How can I host my model on my personal server so that my coworkers can play with it?
- I am not a backend/infrastructure engineer, but I want to make my model available so that it can serve thousands (or even millions) of users. How can I do this at a reasonable price without worrying about scalability and reliability issues?
- There are reasons (such as cost, regulations, privacy, etc.) why I cannot host my model on the cloud, but only on-premises (my work network). Can I serve predictions at scale and reliably in such a case?
- Can I do inference on GPUs?

- How much can I expect to pay for each of these options?
- Could I scale my training and serving across multiple cloud providers?
- How much time and technical know-how will it take to get these running?

Let’s begin our journey by looking at the high-level overview of the tools available to us.

- How much can I expect to pay for each of these options?
- Could I scale my training and serving across multiple cloud providers?
- How much time and technical know-how will it take to get these running?

Let’s begin our journey by looking at the high-level overview of the tools available to us.

Landscape of Serving AI Predictions

There is a multitude of tools, libraries, and cloud services available for getting trained AI models to serve prediction requests. Figure 9-1 simplifies them into four categories.

Figure 9-1. A high-level overview and comparison of different inference serving options

Depending on our inference scenarios, we can make an appropriate choice. Table 9-1 takes a deeper look.

Table 9-1. Tools to serve deep learning models over the network

Category and examples	Expected time to first prediction	Pros and cons
	<5 minutes	

HTTP servers • Flask • Django • Apache OpenWhisk • Python `http.server`		+ Simple to run + Often runs current Python code – Slow – Not optimized for AI
Hosted and managed cloud stacks • Google Cloud ML • Azure ML • Amazon Sage Maker • Algorithmia	<15 minutes	+ Easier GUI/command-line interfaces + Highly scalable + Fully managed, reduces the need for DevOps teams – Usually limited to CPU-based inference, can be slow for large models – Warm-up query time can be slow
Manually managed serving libraries • TensorFlow Serving • NVIDIA TensorRT • DeepDetect • MXNet Model Serving • Skymind Intelligence Layer with DeepLearning4J • Seldon • DeepStack AI Server	<15 minutes	+ High performance + Allows manual controls on optimizations, batching, etc. + Can run inference on GPU – More involved setup – Scaling over multiple nodes usually requires extra groundwork
Cloud AI orchestration frameworks • KubeFlow	~1 hour	+ Makes scaling training and inference easy to manage + Portability between cloud providers + Consistent environments across development and production

	+ For data scientists, integration with familiar tools such as Jupyter Notebooks for sending models to production + Enables composing conditional pipelines to automate testing, cascading models + Uses existing manually managed serving libraries – Still evolving – For beginners, hosted and managed cloud stacks offer an easier learning curve

In this chapter, we explore a range of tools and scenarios. Some of these options are easy to use but limited in functionality. Others offer more granular controls and higher performance but are more involved to set up. We look at one example of each category and take a deeper dive to develop an intuition into when using one of those makes sense. We then present a cost analysis of the different solutions as well as case studies detailing how some of these solutions work in practice today.

	+ For data scientists, integration with familiar tools such as Jupyter Notebooks for sending models to production + Enables composing conditional pipelines to automate testing, cascading models + Uses existing manually managed serving libraries – Still evolving – For beginners, hosted and managed cloud stacks offer an easier learning curve

In this chapter, we explore a range of tools and scenarios. Some of these options are easy to use but limited in functionality. Others offer more granular controls and higher performance but are more involved to set up. We look at one example of each category and take a deeper dive to develop an intuition into when using one of those makes sense. We then present a cost analysis of the different solutions as well as case studies detailing how some of these solutions work in practice today.

Flask: Build Your Own Server

We begin with the most basic technique of *Build Your Own Server* (BYOS). From the choices presented in the first column of Table 9-1, we've selected Flask.

Flask: Build Your Own Server

We begin with the most basic technique of *Build Your Own Server* (BYOS). From the choices presented in the first column of Table 9-1, we've selected Flask.

Making a REST API with Flask

Flask is a Python-based web application framework. Released in 2010 and with more than 46,000 stars on GitHub, it is under continuous development. It's also quick and easy to set up and is really useful for prototyping. It is often the framework of choice for data science practitioners when they want to serve their models to a limited set of users (e.g., sharing with coworkers on a corporate network) without a lot of fuss.

Installing Flask with `pip` is fairly straightforward:

```
$ pip install flask
```

Upon installation, we should be able to run the following simple "Hello World" program:

```
from flask import Flask
app = Flask(__name__)

@app.route("/hello")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

The following is the command to run the "Hello World" program:

```
$ python hello.py
 * Running on http://127.0.0.1:5000/ (Press Ctrl+C to quit)
```

By default, Flask runs on port 5000. When we open the URL *http://localhost:5000/hello* in the browser, we should see the words "Hello World!," as shown in Figure 9-2.

Figure 9-2. Navigate to http://localhost:5000/hello within a web browser to view the "Hello World!" web page

As you can see, it takes barely more than a few lines to get a simple web application up and running. One of the most important lines in that script is `@app.route("/hello")`. It specifies that the path `/hello` after the hostname would be served by the method immediately beneath it. In our case, it merely returns the string "Hello World!" In the next step, we look at how to deploy a Keras model to a Flask server and create a route that will serve predictions by our model.

Figure 9-2. Navigate to http://localhost:5000/hello within a web browser to view the "Hello World!" web page

As you can see, it takes barely more than a few lines to get a simple web application up and running. One of the most important lines in that script is `@app.route("/hello")`. It specifies that the path `/hello` after the hostname would be served by the method immediately beneath it. In our case, it merely returns the string "Hello World!" In the next step, we look at how to deploy a Keras model to a Flask server and create a route that will serve predictions by our model.

Deploying a Keras Model to Flask

Our first step is to load our Keras model. The following lines load the model from the *.h5* file. You'll find the scripts for this chapter on the book's GitHub (see *http://PracticalDeepLearning.ai*) in *code/chapter-9*:

```
from tf.keras.models import load_model
model = load_model("dogcat.h5")
```

Now, we create the route */infer* that would support inference on our images. Naturally, we would support `POST` requests to accept images:

```
@app.route('/infer', methods=[POST])
def infer():
  file = request.files['file']
  image = Image.open(file)
  image = preprocess(image)

  predictions = model.predict(image)
  max_index = numpy.argmax(predictions)
  # We know the labels from the model we trained previously
  if max_index == 0:
     return "Cat"
  else:
     return "Dog"
```

To test the inference, let's use the `curl` command, as follows, on a sample image containing a dog:

```
$ curl -X POST -F image=@dog.jpg 'http://localhost:5000/infer'

{"predictions":[{"label":"dog","probability":0.8525022864341736}]}
```

As expected, we get a prediction of "dog." This has worked quite well so far. At this point, Flask runs only locally; that is, someone else on the network would not be able to make a request to this server. To make Flask available to others, we can simply change `app.run()` to the following:

```
app.run(host="0.0.0.0")
```

At this point, we can give access to our model to anyone within our network. The next question would be—can we do the same to make the model available to the general public? The answer to that question is an emphatic no! The Flask website has a prominent warning stating *"WARNING: Do not use the development server in a production environment."* Flask indeed does not support production work out of the box and would need custom code to enable that. In the upcoming sections, we look at how to host our models on systems that are meant for production use. With all of this in mind, let's recap some of the pros and cons of using Flask.

At this point, we can give access to our model to anyone within our network. The next question would be—can we do the same to make the model available to the general public? The answer to that question is an emphatic no! The Flask website has a prominent warning stating *"WARNING: Do not use the development server in a production environment."* Flask indeed does not support production work out of the box and would need custom code to enable that. In the upcoming sections, we look at how to host our models on systems that are meant for production use. With all of this in mind, let's recap some of the pros and cons of using Flask.

Pros of Using Flask

Flask provides some advantages, namely:

- Quick to set up and to prototype
- Fast development cycle
- Lightweight on resources
- Broad appeal within the Python community

Pros of Using Flask

Flask provides some advantages, namely:

- Quick to set up and to prototype
- Fast development cycle
- Lightweight on resources
- Broad appeal within the Python community

Cons of Using Flask

At the same time, Flask might not be your best choice, for the following reasons

- Cannot scale; by default, it is not meant for production loads. Flask can serve only one request at one time
- Does not handle model versioning out of the box
- Does not support batching of requests out of the box

Cons of Using Flask

At the same time, Flask might not be your best choice, for the following reasons:

- Cannot scale; by default, it is not meant for production loads. Flask can serve only one request at one time
- Does not handle model versioning out of the box
- Does not support batching of requests out of the box

Desirable Qualities in a Production-Level Serving System

For any cloud service that is serving traffic from the public, there are certain attributes that we want to look for when deciding to use a solution. In the contex of machine learning, there are additional qualities that we would look for while building inference services. We look at a few of them if this section.

Desirable Qualities in a Production-Level Serving System

For any cloud service that is serving traffic from the public, there are certain attributes that we want to look for when deciding to use a solution. In the context of machine learning, there are additional qualities that we would look for while building inference services. We look at a few of them if this section.

High Availability

For our users to trust our service, it must be available almost always. For many serious players, they measure their availability metric in terms of "*number of nines*." If a business claims that its service has four 9s availability, they mean the system is up and available 99.99% of the time. Even though 99% sounds impressive, Table 9-2 puts that downtime per year in perspective.

Table 9-2. Downtime per year for different availability percentages

Availability %	Downtime per year
99% ("two nines")	3.65 days
99.9% ("three nines")	8.77 hours
99.99% ("four nines")	52.6 minutes
99.999% ("five nines")	5.26 minutes

Imagine how ridiculous the situation would be if a major website like Amazon were only 99.9% available, losing millions in user revenue during the eight-plus hours of downtime. Five 9s is considered the holy grail. Anything less than three 9s is typically unsuitable for a high-quality production system.

High Availability

For our users to trust our service, it must be available almost always. For many serious players, they measure their availability metric in terms of "*number of nines*." If a business claims that its service has four 9s availability, they mean the system is up and available 99.99% of the time. Even though 99% sounds impressive, Table 9-2 puts that downtime per year in perspective.

Table 9-2. Downtime per year for different availability percentages

Availability %	Downtime per year
99% ("two nines")	3.65 days
99.9% ("three nines")	8.77 hours
99.99% ("four nines")	52.6 minutes
99.999% ("five nines")	5.26 minutes

Imagine how ridiculous the situation would be if a major website like Amazon were only 99.9% available, losing millions in user revenue during the eight-plus hours of downtime. Five 9s is considered the holy grail. Anything less than three 9s is typically unsuitable for a high-quality production system.

Scalability

Traffic handled by production services is almost never uniform across a larger time period. For example, the *New York Times* experiences significantly more traffic during morning hours, whereas Netflix typically experiences a surge in traffic between the evening and late-night hours, when people chill. There are also seasonal factors in traffic. Amazon experiences orders of magnitude more traffic on Black Friday and during Christmas season.

A higher demand requires a higher amount of resources being available and online to serve them. Otherwise, the availability of the system would be in jeopardy. A naive way to accomplish this would be to anticipate the highest volume of traffic the system would ever serve, determine the number of resources necessary to serve that level of traffic, and then allocate that amount all the time, in perpetuity. There are two problems with this approach: 1) if your planning was correct, the resources would be underutilized most of the time, essentially burning money; and 2) if your estimation was insufficient, you might end up affecting the availability of your service and end up with a far worse problem of losing the trust of your customers and ultimately their wallets.

A smarter way to manage traffic loads is to monitor them as they are coming in and dynamically allocate and deallocate resources that are available for service. This ensures that the increased traffic is handled without loss of service while keeping operating costs to a minimum during low-traffic times.

When scaling down resources, any resource that is about to be deallocated is quite likely to be processing traffic at that moment. It’s essential to ensure that all of those requests be completed before shutting down that resource. Also, crucially, the resource must not process any new requests. This process is called *draining*. Draining is also crucial when machines are taken down for routine maintenance and/or upgrades.

Scalability

Traffic handled by production services is almost never uniform across a larger time period. For example, the *New York Times* experiences significantly more traffic during morning hours, whereas Netflix typically experiences a surge in traffic between the evening and late-night hours, when people chill. There are also seasonal factors in traffic. Amazon experiences orders of magnitude more traffic on Black Friday and during Christmas season.

A higher demand requires a higher amount of resources being available and online to serve them. Otherwise, the availability of the system would be in jeopardy. A naive way to accomplish this would be to anticipate the highest volume of traffic the system would ever serve, determine the number of resources necessary to serve that level of traffic, and then allocate that amount all the time, in perpetuity. There are two problems with this approach: 1) if your planning was correct, the resources would be underutilized most of the time, essentially burning money; and 2) if your estimation was insufficient, you might end up affecting the availability of your service and end up with a far worse problem of losing the trust of your customers and ultimately their wallets.

A smarter way to manage traffic loads is to monitor them as they are coming in and dynamically allocate and deallocate resources that are available for service. This ensures that the increased traffic is handled without loss of service while keeping operating costs to a minimum during low-traffic times.

When scaling down resources, any resource that is about to be deallocated is quite likely to be processing traffic at that moment. It’s essential to ensure that all of those requests be completed before shutting down that resource. Also, crucially, the resource must not process any new requests. This process is called *draining*. Draining is also crucial when machines are taken down for routine maintenance and/or upgrades.

Low Latency

Consider these facts. Amazon published a study in 2008 in which it found that every 100 ms increase in latency in its retail website resulted in a 1% loss of profit. A one-second delay in loading the website caused a whopping $1.6 billion in lost revenue! Google found that a 500 ms latency on mobile websites resulted in a traffic drop of 20%. In other words, a 20% decrease in the opportunity-to-serve advertisements. And this does not affect only industry giants. If a web page takes longer than three seconds to load on a mobile phone, 53% of users abandon it (according to a 2017 study by Google). It's clear that time is money.

Reporting average latency can be misleading because it might paint a cheerier picture than a ground reality. It's like saying if Bill Gates walks into a room, everyone is a billionaire on average. Instead, percentile latency is the typically reported metric. For example, a service might report 987 ms @ 99th percentile. This means that 99% of the requests were served in 987 ms or less. The very same system could have a 20 ms latency on average. Of course, as traffic to your service increases, the latency might increase if the service is not scaled up to give adequate resources. As such, latency, high availability, and scalability are intertwined.

Low Latency

Consider these facts. Amazon published a study in 2008 in which it found that every 100 ms increase in latency in its retail website resulted in a 1% loss of profit. A one-second delay in loading the website caused a whopping $1.6 billion in lost revenue! Google found that a 500 ms latency on mobile websites resulted in a traffic drop of 20%. In other words, a 20% decrease in the opportunity-to-serve advertisements. And this does not affect only industry giants. If a web page takes longer than three seconds to load on a mobile phone, 53% of users abandon it (according to a 2017 study by Google). It's clear that time is money.

Reporting average latency can be misleading because it might paint a cheerier picture than a ground reality. It's like saying if Bill Gates walks into a room, everyone is a billionaire on average. Instead, percentile latency is the typically reported metric. For example, a service might report 987 ms @ 99th percentile. This means that 99% of the requests were served in 987 ms or less. The very same system could have a 20 ms latency on average. Of course, as traffic to your service increases, the latency might increase if the service is not scaled up to give adequate resources. As such, latency, high availability, and scalability are intertwined.

Geographic Availability

The distance between New York and Sydney is nearly 10,000 miles (16,000 km) The speed of light in a vacuum is roughly 186,282 miles per second (300,000 kn per second). Silica glass (used in fiber-optic cables) decreases the speed of light by a factor of roughly 30% down to 130,487 miles per second (210,000 km per second). On a piece of fiber-optic running in a straight line between these two cities, the roundtrip travel time alone for a single request is nearly 152 ms. Keep in mind that this does not account for the amount of time it takes for the request to be processed at the server, or the hops that the packets need to make across multiple routers along the way. This level of service would be unacceptable for many applications.

Services that expect to be used throughout the world must be strategically located to minimize latency for the users in those regions. Additionally, resources can be dynamically scaled up or down depending on local traffic, thus giving more granular control. The major cloud providers have a presence on at least five continents (sorry penguins!).

TIP

Want to simulate how long the incoming requests would take from your computer to a particular datacenter around the world? Table 9-3 lists a few handy browser-based tools offered by cloud providers.

Table 9-3. Latency measurement tools for different cloud providers

Service	Cloud provider
AzureSpeed.com	Microsoft Azure
CloudPing.info	Amazon Web Services
GCPing.com	Google Cloud Platform

Additionally, to determine realistic combinations of latency from one location to another, *CloudPing.co* measures AWS Inter-Region Latency, between more than 16 US-based AWS datacenters to one another.

.
ɹ

Failure Handling

There's an old saying that there are only two things that are assured in life—death and taxes. In the twenty-first century, this adage applies not just to humans but also computer hardware. Machines fail all the time. The question is never *if* a machine will fail, it's *when*. One of the necessary qualities of production-quality service is its ability to gracefully handle failures. If a machine goes down, quickly bring up another machine to take its place and continue serving traffic. I an entire datacenter goes down, seamlessly route traffic to another datacenter so that users don't even realize that anything bad happened in the first place.

Failure Handling

There’s an old saying that there are only two things that are assured in life—death and taxes. In the twenty-first century, this adage applies not just to humans but also computer hardware. Machines fail all the time. The question is never *if* a machine will fail, it’s *when*. One of the necessary qualities of production-quality service is its ability to gracefully handle failures. If a machine goes down, quickly bring up another machine to take its place and continue serving traffic. If an entire datacenter goes down, seamlessly route traffic to another datacenter so that users don’t even realize that anything bad happened in the first place.

Monitoring

If you can’t measure it, you can’t improve it. Worse, does it even exist? Monitoring the number of requests, availability, latency, resource usage, number of nodes, distribution of traffic, and location of users is vital to understanding how a service performs; finding opportunities to improve it; and, more importantly, how much to pay. Most cloud providers already have built-in dashboards providing these metrics. Additionally, recording task-specific analytics like time for model inference, preprocessing, and so on can add another level of understanding.

Monitoring

If you can’t measure it, you can’t improve it. Worse, does it even exist? Monitoring the number of requests, availability, latency, resource usage, number of nodes, distribution of traffic, and location of users is vital to understanding how a service performs; finding opportunities to improve it; and, more importantly, how much to pay. Most cloud providers already have built-in dashboards providing these metrics. Additionally, recording task-specific analytics like time for model inference, preprocessing, and so on can add another level of understanding.

Model Versioning

We have learned in this book (and will continue to learn all the way to the last page) that machine learning is always iterative. Particularly in the case of applications in the real world, data that the model can learn on is constantly being generated. Moreover, the incoming data distribution might shift over time compared to what it was trained on, leading to lower prediction power (a phenomenon called c*oncept drift*). To provide users with the best possible experience, we want to keep improving our models. Every time we train our model with newer data to further improve its accuracy and make the best version yet, we want to make it available for our users as quickly and seamlessly as possible. Any good production-quality inference system should provide the ability to provide different versions of a model, including the ability to swap a live version of the model with another version at a moment's notice.

Model Versioning

We have learned in this book (and will continue to learn all the way to the last page) that machine learning is always iterative. Particularly in the case of applications in the real world, data that the model can learn on is constantly being generated. Moreover, the incoming data distribution might shift over time compared to what it was trained on, leading to lower prediction power (a phenomenon called c*oncept drift*). To provide users with the best possible experience, we want to keep improving our models. Every time we train our model with newer data to further improve its accuracy and make the best version yet, we want to make it available for our users as quickly and seamlessly as possible. Any good production-quality inference system should provide the ability to provide different versions of a model, including the ability to swap a live version of the model with another version at a moment's notice.

A/B Testing

In addition to supporting multiple versions of a model, there are reasons we'd want to serve different versions of the model at the same time depending on a variety of attributes such as the geographic location of the user, demographics, o simply by random assignment.

A/B testing is a particularly useful tool when improving a model. After all, if our spanking new model was flawed in some way, we'd rather it be deployed to only a small subset of our users rather than 100% of them at the same time before we catch the flaw. Additionally, if a model meets criteria for success on that small subset, it provides validation for the experiment and justifies eventually being promoted to all users.

A/B Testing

In addition to supporting multiple versions of a model, there are reasons we'd want to serve different versions of the model at the same time depending on a variety of attributes such as the geographic location of the user, demographics, or simply by random assignment.

A/B testing is a particularly useful tool when improving a model. After all, if our spanking new model was flawed in some way, we'd rather it be deployed to only a small subset of our users rather than 100% of them at the same time before we catch the flaw. Additionally, if a model meets criteria for success on that small subset, it provides validation for the experiment and justifies eventually being promoted to all users.

Support for Multiple Machine Learning Libraries

Last but not least, we don’t want to be locked into a single machine learning library. Some data scientists in an organization might train models in PyTorch, others in TensorFlow, or maybe scikit-Learn suffices for non–deep learning tasks. The flexibility to support multiple libraries would be a welcome bonus.

Support for Multiple Machine Learning Libraries

Last but not least, we don’t want to be locked into a single machine learning library. Some data scientists in an organization might train models in PyTorch, others in TensorFlow, or maybe scikit-Learn suffices for non–deep learning tasks. The flexibility to support multiple libraries would be a welcome bonus.

Google Cloud ML Engine: A Managed Cloud AI Serving Stack

Considering all the desirable qualities we discussed in a production environment in the previous section, it's generally not a good idea to use Flask for serving users. If you do not have a dedicated infrastructure team and would like to spend more time making better models than deploying them, using a managed cloud solution is the right approach. There are several cloud-based Inference-as-a-Service solutions on the market today. We have chosen to explore the Google Cloud ML Engine partly because of the convenient TensorFlow integration and partly because it ties in nicely with the ML Kit material that we touch upon in Chapter 13.

Google Cloud ML Engine: A Managed Cloud AI Serving Stack

Considering all the desirable qualities we discussed in a production environment in the previous section, it's generally not a good idea to use Flask for serving users. If you do not have a dedicated infrastructure team and would like to spend more time making better models than deploying them, using a managed cloud solution is the right approach. There are several cloud-based Inference-as-a-Service solutions on the market today. We have chosen to explore the Google Cloud ML Engine partly because of the convenient TensorFlow integration and partly because it ties in nicely with the ML Kit material that we touch upon in Chapter 13.

Pros of Using Cloud ML Engine

- Easy-to-deploy models in production with web-based GUI
- Powerful and easily scalable to millions of users
- Provides deep insights into model usage
- Ability to version models

Pros of Using Cloud ML Engine

- Easy-to-deploy models in production with web-based GUI
- Powerful and easily scalable to millions of users
- Provides deep insights into model usage
- Ability to version models

Cons of Using Cloud ML Engine

- High latency, offers only CPUs for inference (as of August 2019)
- Unsuitable for scenarios involving legal and data privacy issues where the data must not leave the network
- Imposes restrictions on architecture design of complex applications

Cons of Using Cloud ML Engine

- High latency, offers only CPUs for inference (as of August 2019)
- Unsuitable for scenarios involving legal and data privacy issues where the data must not leave the network
- Imposes restrictions on architecture design of complex applications

Building a Classification API

The following step-by-step guide shows how to go about uploading and hosting our a Dog/Cat classifier model on Google Cloud ML Engine:

1. Create a model on the Google Cloud ML Engine dashboard at *https://console.cloud.google.com/mlengine/models*. Because this is the first time we're using the dashboard, we need to click ENABLE API, as depicted in Figure 9-3.

Figure 9-3. Listing page for machine learning models on the Google Cloud ML Engine dashboard

2. Give the model a name and a description (Figure 9-4).

Figure 9-4. Model creation page on Google Cloud ML Engine

3. After the model is created, we can access the model on the listing page (Figure 9-5).

Figure 9-5. Model listings page on Google Cloud ML Engine

4. Click the model to go to the model details page (Figure 9-6) and add a new version.

Figure 9-6. Details page of the just-created Dog/Cat classifier

5. Fill out the necessary information to create the new version. The last field a the bottom requires you to upload the model to Google Cloud Storage before we can use it. Click the Browse button to create a new bucket for storing the model (Figure 9-7).

t

Figure 9-7. Creating a new version for a machine learning model

6. Create a new bucket with a unique name, a storage class, and region. After you create this bucket, go to

https://console.cloud.google.com/storage/browser (in a separate tab while keeping the current one open) to find this newly created bucket and upload the model there (Figure 9-8).

Figure 9-8. Creating a new Google Cloud Storage bucket within the ML model version creation pag

7. Our Dog/Cat classifier model is an *.h5* file. However, Google Cloud expect a SavedModel file. You can find the script to convert the *.h5* file to SavedModel on the book's GitHub repository (see http://PracticalDeepLearning.ai) at *code/chapter-9/scripts/h5_to_tf.ipynb*.

Simply load the model and execute the rest of the notebook.

8. In the Google Cloud Storage browser, upload the newly converted model (Figure 9-9) to the bucket you created in step 6.

Figure 9-9. Google Cloud Storage Browser page showing the uploaded Dog/Cat classifier model in TensorFlow format

9. Specify the URI on the model version creation page for the model that you just uploaded (Figure 9-10).

Figure 9-10. Add the URI for the model you uploaded to Google Cloud Storage

10. Click the Save button and wait for the model version to be created. As soon as the model version is created, you can begin making predictions against it

11. If it's not already present on your machine, you can download and install the Google Cloud SDK from the installation website at *https://cloud.google.com/sdk/install*.

12. You can use the Cloud ML Engine REST API to make your requests. However, for brevity, use the command-line tools in the Cloud SDK. You first need to convert your image into a *request.json* file using the *image-to-json.py* script located at *code/chapter-9*:

```
$ python image-to-json.py --input dog.jpg --output request.json
```

13. Next, use the *request.json* file created in the previous step to execute a request against our model:

```
$ time cloud ai-platform predict --model DogCat --version v1
                                 --json-instances
request.json

SCORES
[0.14749771356, 0.8525022864]

real     0m3.370s
user     0m0.811s
sys 0m0.182s
```

As you can see from the output, we get similar results as with our Flask server; that is, a prediction of “dog” with 85% confidence.

NOTE

If this is your first time using **`gcloud`**, you need to run the following command to tie the command-line tool to your Google account:

```
$ gcloud auth login
```

Next, select the project using the following command:

```
$ gcloud config set project {project_name}
```

Piece of cake, wasn't it? In our example, we used the Google Cloud SDK to request a prediction for the sake of brevity. In a production scenario, you would want to execute the same prediction request using Google's API endpoints, instead; either by generating HTTP requests or by using their client libraries. We can follow the documentation on Google Cloud Docs for production scenarios.

At this point, the model is ready to be served to any user anywhere in the world using applications on the browser, mobile and edge devices, desktop, as well as cloud environments. Using a hosted stack is a pretty viable option for individuals and organizations who want the flexibility and reliability that the cloud provides while having to do minimal setup and maintenance work for the infrastructure.

In contrast, there are situations for which a hosted solution might not be the best approach. Reasons could include pricing models, data privacy issues, legal questions, technical issues, trust concerns, or contractual obligations. In such cases, a solution that is hosted and managed locally (or, "on-premises") would be preferable.

TIP

For processing a large number of images at one time, you can modify *image-to-json.py* to create a *request.json* file that contains an array of multiple inputs.

Piece of cake, wasn't it? In our example, we used the Google Cloud SDK to request a prediction for the sake of brevity. In a production scenario, you would want to execute the same prediction request using Google's API endpoints, instead; either by generating HTTP requests or by using their client libraries. We can follow the documentation on Google Cloud Docs for production scenarios.

At this point, the model is ready to be served to any user anywhere in the world using applications on the browser, mobile and edge devices, desktop, as well as cloud environments. Using a hosted stack is a pretty viable option for individuals and organizations who want the flexibility and reliability that the cloud provides while having to do minimal setup and maintenance work for the infrastructure.

In contrast, there are situations for which a hosted solution might not be the best approach. Reasons could include pricing models, data privacy issues, legal questions, technical issues, trust concerns, or contractual obligations. In such cases, a solution that is hosted and managed locally (or, "on-premises") would be preferable.

TIP

For processing a large number of images at one time, you can modify *image-to-json.py* to create a *request.json* file that contains an array of multiple inputs.

TensorFlow Serving

TensorFlow Serving is an open source library in the TensorFlow ecosystem for serving machine learning models fast. Unlike Flask, it's built for performance, with low overhead, and designed for use in production. TensorFlow Serving is widely used by large companies to serve their models for prediction services. It is one of the integral components of TensorFlow Extended (TFX)—an end-to-end deep learning pipeline in the TensorFlow ecosystem.

As we saw when we looked at the desired qualities of a production system, TensorFlow serving offers low latency, failure handling, high throughput, and model versioning. Another benefit includes the ability to serve multiple models at the same time on the same service. It implements several techniques to speed up serving:

- During server startup, it starts a burst of threads for fast model loading.
- It uses separate thread pools for loading models and for serving inferences while giving higher priority to the threads in the inference pool. This is crucial to lowering request latency.
- It builds minibatches of incoming asynchronous requests for short periods of time. As we have seen, with the power of batching data on GPUs during training, it aims to bring similar efficiencies on asynchronous requests. As an example, waiting 500 ms to group together several requests for inference. While at worst case, this adds a 500 ms penalty for the first request in the batch, it reduces the average latency across requests and maximizes hardware utilization.

NOTE

TensorFlow serving gives you full control over the model rollout procedure. You can serve different models or different versions of the same kind of model in the same process. You just need to make sure that you know the name and location of the version that you want to remove or put into production.

Installation

There are a few different ways of setting up TensorFlow Serving:

- Building from source
- Downloading and installing using APT
- Deploying Docker images

If you're feeling adventurous, building from source might be the danger you seek. But if you just want to get up and running quickly, we recommend using Docker because it requires minimal steps to get the system up and running. Wha is Docker, you might ask? Docker provides virtualization of a Linux environment for applications that run within it. It provides isolation of resources that essentially operate as a clean slate for setting up an environment in which ar application can run. Typically, an application and all of its dependencies are packaged into a single Docker container that can then be deployed repeatedly as necessary. Because the application is set up in a clean environment, it reduces the likelihood of configuration and deployment errors. This makes Docker very well suited for running applications in production.

The biggest benefit that Docker provides for us is alleviating "dependency hell" because all the necessary dependencies are packaged within the container. One additional advantage of using Docker is that the process of setting up your application remains more or less the same across different platforms, whether you use Windows, Linux, or Mac.

The Docker installation instructions, depending on the target platform, are available on the Docker home page. This should not take more than a few minutes because the setup is fairly straightforward. After you've installed Docker, you can run the following command to set up TensorFlow Serving for CPU:

```
$ docker run -p 8501:8501 \
--mount type=bind,source=/path/to/dogcat/,target=/models/dogcat \
-e MODEL_NAME=dogcat -t tensorflow/serving
```

For GPU-enabled machines, run the following command, instead:

```
$ docker run -p 8501:8501 --runtime=nvidia \
--mount type=bind,source=/path/to/dogcat/,target=/models/dogcat \
-e MODEL_NAME=dogcat -t tensorflow/serving
```

In either case, if everything went smoothly, you should have a REST API running on your local port 8501 serving our Dog/Cat classifier.

NOTE

In any inference request, the end-to-end latency is a summation of time taken by multiple steps along the process. This includes round-trip network time, request time to serialize/deserialize the request and response objects, and, of course, time to perform the actual inference. One more component that adds overhead is the serving framework; that is, TensorFlow Serving. Google claims that the overhead contributed by TensorFlow Serving is minimal. In its experiments, it observed that TensorFlow Serving alone was able to handle approximately 100,000 QPS per core on a 16 vCPU Intel Xeon E5 2.6 GHz machine. Because it is measuring the overhead, this excludes the remote procedure call (RPC) time and the TensorFlow inference processing time.

Even though TensorFlow Serving is a great choice for serving inferences from a single machine, it does not have built-in functionality for horizontal scaling. Instead, it is built to be used in conjunction with other systems that can supercharge TensorFlow Serving with dynamic scaling. We explore one such solution in the following section.

```
$ docker run -p 8501:8501 --runtime=nvidia \
--mount type=bind,source=/path/to/dogcat/,target=/models/dogcat \
-e MODEL_NAME=dogcat -t tensorflow/serving
```

In either case, if everything went smoothly, you should have a REST API running on your local port 8501 serving our Dog/Cat classifier.

NOTE

In any inference request, the end-to-end latency is a summation of time taken by multiple steps along the process. This includes round-trip network time, request time to serialize/deserialize the request and response objects, and, of course, time to perform the actual inference. One more component that adds overhead is the serving framework; that is, TensorFlow Serving. Google claims that the overhead contributed by TensorFlow Serving is minimal. In its experiments, it observed that TensorFlow Serving alone was able to handle approximately 100,000 QPS per core on a 16 vCPU Intel Xeon E5 2.6 GHz machine. Because it is measuring the overhead, this excludes the remote procedure call (RPC) time and the TensorFlow inference processing time.

Even though TensorFlow Serving is a great choice for serving inferences from a single machine, it does not have built-in functionality for horizontal scaling. Instead, it is built to be used in conjunction with other systems that can supercharge TensorFlow Serving with dynamic scaling. We explore one such solution in the following section.

KubeFlow

Throughout this book, we have explored the various steps of an end-to-end deep learning pipeline, from data ingestion, analysis, distributed training (including hyperparameter tuning) at scale, tracking experiments, deployment, and eventually to serving prediction requests at scale. Each of these steps is complex in its own right, with its set of tools, ecosystems, and areas of expertise. People dedicate their lifetimes developing expertise in just one of these fields. It's not exactly a walk in the park. The combinatorial explosion of the know-how required when factoring for the necessary backend engineering, hardware engineering, infrastructure engineering, dependency management, DevOps, fault tolerance, and other engineering challenges can result in a very expensive hiring process for most organizations.

As we saw in the previous section, Docker saves us the hassle of dependency management by making portable containers available. It helps us make TensorFlow Serving available across platforms easily without having to build it from source code or install dependencies manually. Great! But it still doesn't have an answer to many of the other challenges. How are we going to scale up containers to match rises in demand? How would we efficiently distribute traffic across containers? How do we ensure that the containers are visible to one another and can communicate?

These are questions answered by *Kubernetes*. Kubernetes is an orchestration framework for automatically deploying, scaling, and managing containers (like Docker). Because it takes advantage of the portability offered by Docker, we can use Kubernetes to deploy to developer laptops as well as thousand-machine clusters in an almost identical manner. This helps maintain consistency across different environments, with the added benefit of scalability in an accessible manner. It is worth noting that Kubernetes is not a dedicated solution for machine learning (neither is Docker); rather, it is a general-purpose solution to many of the problems faced in software development, which we use in the context of deep learning.

But let's not get ahead of ourselves just yet. After all, if Kubernetes were the be-all and end-all solution, it would have appeared in the chapter title! A machine

learning practitioner using Kubernetes still needs to assemble all of the appropriate sets of containers (for training, deployment, monitoring, API management, etc.) that then need to be orchestrated together to make a fully functioning end-to-end pipeline. Unfortunately, many data scientists are trying to do exactly this in their own silos, reinventing the wheel building ad hoc machine learning-specific pipelines. Couldn't we save everyone the trouble and make one Kubernetes-based solution for machine learning scenarios?

Enter *KubeFlow*, which promises to automate a large chunk of these engineering challenges and hide the complexity of running a distributed, scalable, end-to-end deep learning system behind a web GUI-based tool and a powerful command-line tool. This is more than just an inference service. Think of it as a large ecosystem of tools that can interoperate seamlessly and, more importantly, scale up with demand. KubeFlow is built for the cloud. Though not just one cloud—it's built to be compatible with all major cloud providers. This has significant implications on cost. Because we are not tied to a specific cloud provider, we have the freedom to move all of our operations at a moment's notice if a competing cloud provider drops its prices. After all, competition benefits consumers.

KubeFlow supports a variety of hardware infrastructure, from developer laptops and on-premises datacenters, all the way to public cloud services. And because it's built on top of Docker and Kubernetes, we can rest assured that the environments will be identical whether deployed on a developer laptop or a large cluster in a datacenter. Every single way in which the developer setup is different from the production environment could result in an outage, so it's really valuable to have this consistency across environments.

Table 9-4 shows a brief list of readily available tools within the KubeFlow ecosystem.

Table 9-4. Tools available on KubeFlow

Tool	Functionality
Jupyter Hub	Notebook environment
TFJob	Training TensorFlow models
TensorFlow Serving	Serving TensorFlow models

Seldon	Serving models
NVIDIA TensorRT	Serving models
Intel OpenVINO	Serving models
KFServing	Abstraction for serving Tensorflow, XGBoost, scikit-learn, PyTorch, and ONNX models
Katib	Hyperparameter tuning and NAS
Kubebench	Running benchmarking jobs
PyTorch	Training PyTorch models
Istio	API services, authentication, A/B testing, rollouts, metrics
Locust	Load testing
Pipelines	Managing experiments, jobs, and runs, scheduling machine learning workflows

As the joke goes in the community, with so many technologies prepackaged, KubeFlow finally makes our résumés buzzword- (and recruiter-) compliant.

NOTE

Many people assume that KubeFlow is a combination of Kubernetes and TensorFlow, which, as you have seen, is not the case. It is that and much more.

There are two important parts to KubeFlow that make it unique: pipelines and fairing.

Seldon	Serving models
NVIDIA TensorRT	Serving models
Intel OpenVINO	Serving models
KFServing	Abstraction for serving Tensorflow, XGBoost, scikit-learn, PyTorch, and ONNX models
Katib	Hyperparameter tuning and NAS
Kubebench	Running benchmarking jobs
PyTorch	Training PyTorch models
Istio	API services, authentication, A/B testing, rollouts, metrics
Locust	Load testing
Pipelines	Managing experiments, jobs, and runs, scheduling machine learning workflows

As the joke goes in the community, with so many technologies prepackaged, KubeFlow finally makes our résumés buzzword- (and recruiter-) compliant.

NOTE

Many people assume that KubeFlow is a combination of Kubernetes and TensorFlow, which, as you have seen, is not the case. It is that and much more.

There are two important parts to KubeFlow that make it unique: pipelines and fairing.

Pipelines

Pipelines give us the ability to compose steps across the machine learning to schedule complex workflows. Figure 9-11 shows us an example of a pipeline. Having visibility into the pipeline through a GUI tool helps stakeholders understand it (beyond just the engineers who built it).

Figure 9-11. An end-to-end pipeline illustrated in KubeFlow

Pipelines

Pipelines give us the ability to compose steps across the machine learning to schedule complex workflows. Figure 9-11 shows us an example of a pipeline. Having visibility into the pipeline through a GUI tool helps stakeholders understand it (beyond just the engineers who built it).

Figure 9-11. An end-to-end pipeline illustrated in KubeFlow

Fairing

Fairing allows us to manage the entire build, train, and deploy lifecycle directly through Jupyter Notebooks. Figure 9-12 shows how to start a new notebook server, where we can host all of our Jupyter Notebooks, run training on them, and deploy our models to Google Cloud using the following few lines of code, all the while being in the comfort of a very familiar Jupyter environment:

```
from fairing.deployers.gcp.gcpserving import GCPServingDeployer
GCPServingDeployer().deploy(model_dir, model_name, version_name)
```

Figure 9-12. Creating a new Jupyter Notebook server on KubeFlow

Installation

Creating a new KubeFlow deployment is a fairly straightforward process that is well documented on the KubeFlow website. You can set up KubeFlow using the browser for GCP. Alternatively, you can use the KubeFlow command-line tool to set up a deployment on GCP, AWS, and Microsoft Azure. Figure 9-13 shows a GCP deployment using the web browser.

Figure 9-13. Creating a KubeFlow deployment on GCP using the browser

As of this writing, KubeFlow is in active development and shows no signs of

stopping. Companies such as Red Hat, Cisco, Dell, Uber, and Alibaba are some of the active contributors on top of cloud giants like Microsoft, Google, and IBM. Ease and accessibility for solving tough challenges attract more people to any platform, and KubeFlow is doing exactly that.

stopping. Companies such as Red Hat, Cisco, Dell, Uber, and Alibaba are some of the active contributors on top of cloud giants like Microsoft, Google, and IBM. Ease and accessibility for solving tough challenges attract more people to any platform, and KubeFlow is doing exactly that.

Price Versus Performance Considerations

In Chapter 6, we looked at how to improve our model performance for inference (whether on smartphones or on a server). Now let's look from another side: the hardware performance and the price involved.

Often while building a production system, we want the flexibility of choosing suitable hardware to strike the proper balance between performance, scale, and price for our scenario. Consider building an app that requires cloud-based inference. We can go set up our own stack manually (using Flask or TensorFlow Serving or KubeFlow), or we could use a managed Inference-as-a-Service stack (like the Google Cloud ML Engine). Assuming that our service went viral, let's see how much it would cost.

Price Versus Performance Considerations

In Chapter 6, we looked at how to improve our model performance for inference (whether on smartphones or on a server). Now let's look from another side: the hardware performance and the price involved.

Often while building a production system, we want the flexibility of choosing suitable hardware to strike the proper balance between performance, scale, and price for our scenario. Consider building an app that requires cloud-based inference. We can go set up our own stack manually (using Flask or TensorFlow Serving or KubeFlow), or we could use a managed Inference-as-a-Service stack (like the Google Cloud ML Engine). Assuming that our service went viral, let's see how much it would cost.

Cost Analysis of Inference-as-a-Service

For Google Cloud ML Engine, as of August 2019 in North America, it costs a rather inexpensive $0.0401 per hour of combined inference time on a single-core CPU machine. There's also an option for a quad-core CPU machine, but really, a single core should suffice for most applications. Running several queries to the server with a small image of 12 KB took roughly 3.5 seconds on average, as illustrated in Figure 9-14. This does sound slow, and is partly because of doing inference on a moderate-speed machine, and, more important on a CPU server. It's worth mentioning that this benchmark is on a warmed-up machine that has recently received an API request and hence has the model preloaded. For comparison, the first query takes between 30 and 60 seconds. This shows the importance of keeping the service running constantly or sending frequent warm-up queries. This happens because the Google Cloud ML engine takes down a model if it notices a prolonged period of nonuse.

Figure 9-14. Google Cloud ML Engine showing incoming queries and latency of serving the calls, with end-to-end latency at user's end of about 3.5 seconds

If a request came in at every second for an entire month, there would be a total

of 60 x 60 x 24 x 30 = 2,592,000 calls per month. Assuming that each inference takes 3.5 seconds, a single node would be insufficient. The cloud service would quickly realize that and, in response to the increased traffic, bring up three additional machines to handle the traffic. In total, with four machines running fo a month at $0.0401 per hour per node, it would cost a grand total of $115.48. To put this into perspective, for two million calls, that's about the cost of a cup of Starbucks coffee a day for an entire month. And let's not forget this is without involving much of the DevOps team members, whose time is expensive. If we took the hypothetical scenario of a Yelp-like service for which users, on average, upload photos of food at 64 QPS, running inferences on them using a classification model would cost only $7,390.

of 60 x 60 x 24 x 30 = 2,592,000 calls per month. Assuming that each inference takes 3.5 seconds, a single node would be insufficient. The cloud service would quickly realize that and, in response to the increased traffic, bring up three additional machines to handle the traffic. In total, with four machines running for a month at $0.0401 per hour per node, it would cost a grand total of $115.48. To put this into perspective, for two million calls, that's about the cost of a cup of Starbucks coffee a day for an entire month. And let's not forget this is without involving much of the DevOps team members, whose time is expensive. If we took the hypothetical scenario of a Yelp-like service for which users, on average, upload photos of food at 64 QPS, running inferences on them using a classification model would cost only $7,390.

Cost Analysis of Building Your Own Stack

Less spending and high scalability, now that's a winning combination. But the one downside is the total roundtrip latency of each request. Taking matters into our own hands, getting a VM with a modest GPU on the cloud, and setting up our scaling pipeline (using KubeFlow or the native cloud load-balancing features with TensorFlow Serving), we would be able to respond either in milliseconds or batch a few incoming queries together (say every 500 ms) to serve them. As an example, looking at the inventory of VMs on Azure, for $2.07 per hour, we can rent out an ND6 machine that features an NVIDIA P40 GPU and 112 GiB RAM By batching incoming requests every 500 ms to 1 second, this machine can serve 64 requests per second at a total cost of $1,490 per month, and faster than the Google Cloud ML Engine.

In summary, the cost savings and performance benefits of orchestrating our own cloud machine environment kicks in big time when working on large QPS scenarios, as demonstrated in Figure 9-15.

Figure 9-15. Cost comparison of infrastructure as a service (Google Cloud ML Engine) versus building your own stack over virtual machines (Azure VM) (costs as of August 2019)

TIP

A common question that arises while benchmarking is what is my system's limit? JMeter can help answer this. JMeter is a load-testing tool that lets you perform stress testing of your system with an easy-

to-use graphical interface. It lets you create reusable configurations to simulate a variety of usage scenarios.

s

r

.

e

to-use graphical interface. It lets you create reusable configurations to simulate a variety of usage scenarios.

Summary

In this chapter, we answered the question most engineers and developers ask: how do we serve model prediction requests at scale for applications in the real world? We explored four different methods of serving an image recognition model: using Flask, Google Cloud ML, TensorFlow Serving, and KubeFlow. Depending on the scale, latency requirements, and our skill level, some solutions might be more attractive than others. Finally, we developed an intuition into the cost effectiveness of different stacks. Now that we can show our fabulous classifier model off to the world, all that's left is to make our work go viral!

Summary

In this chapter, we answered the question most engineers and developers ask: how do we serve model prediction requests at scale for applications in the real world? We explored four different methods of serving an image recognition model: using Flask, Google Cloud ML, TensorFlow Serving, and KubeFlow. Depending on the scale, latency requirements, and our skill level, some solutions might be more attractive than others. Finally, we developed an intuition into the cost effectiveness of different stacks. Now that we can show our fabulous classifier model off to the world, all that's left is to make our work go viral!

Chapter 10. AI in the Browser with TensorFlow.js and ml5.js

Written in collaboration with guest author: Zaid Alyafeai

You're a developer who dreams big. You have a kickass AI model that you would like a lot of people to try. How many is a lot? Ten thousand? A million? No, silly. You like to dream big. How about 100 million people? That's a nice round number. Now convincing 100 million people to download and install an app and make space for it on their phones is not an easy sell. But what if we told you that they all have an app already installed, just for you. No downloads. No installs. No app stores. What kind of black magic is this!? Of course, it's the web browser. And as a bonus, it also runs on your PC.

This is what Google did with its home page when it decided to launch its first-ever AI doodle to their billions of users (Figure 10-1). And what better theme to pick for it than the music of J.S. Bach. (Bach's parents wanted to call him J.S. Bach, 310 years before JavaScript was even created. They had quite the foresight!)

To explain briefly, the doodle allowed anyone to write one line (voice) of random notes for two measures using mouse clicks. When the user clicked a button labeled Harmonize, the input would then be processed against hundreds of musical pieces written by Bach that contain between two and four lines (voices) of music. The system would figure out which notes would sound best along with the user's input to create a much richer Bach-like sounding musical piece. The entire process ran in the browser, so Google would not need to scale up its machine learning prediction infrastructure at all.

Figure 10-1. The Bach music harmonizer doodle from Google

In addition to the cost savings and the ability to run on any platform, with a browser we can provide users with a richer, more interactive experience because network latency is not a factor. And of course, because everything can be run locally after the model is downloaded, the end user can benefit from the privacy of their data.

Given that JavaScript is the language of the web browser, it's useful for us to delve into JavaScript-based deep learning libraries that can run our trained model within users' browsers. And that's exactly what we do in this chapter.

Here, we focus on implementing deep learning models in the browser. First, we look at a brief history of different JavaScript-based deep learning frameworks before moving on to TensorFlow.js and eventually a higher-level abstraction for it called ml5.js. We also examine a few complex browser-based applications such as detecting the body pose of a person or converting a hand-drawn doodle to a photograph (using GANs). Finally, we talk about some practical considerations and showcase some real-world case studies.

Figure 10-1. The Bach music harmonizer doodle from Google

In addition to the cost savings and the ability to run on any platform, with a browser we can provide users with a richer, more interactive experience because network latency is not a factor. And of course, because everything can be run locally after the model is downloaded, the end user can benefit from the privacy of their data.

Given that JavaScript is the language of the web browser, it's useful for us to delve into JavaScript-based deep learning libraries that can run our trained model within users' browsers. And that's exactly what we do in this chapter.

Here, we focus on implementing deep learning models in the browser. First, we look at a brief history of different JavaScript-based deep learning frameworks before moving on to TensorFlow.js and eventually a higher-level abstraction for it called ml5.js. We also examine a few complex browser-based applications such as detecting the body pose of a person or converting a hand-drawn doodle to a photograph (using GANs). Finally, we talk about some practical considerations and showcase some real-world case studies.

JavaScript-Based Machine Learning Libraries: A Brief History

Since the breakthrough of deep learning in recent years, many attempts have been made to make AI accessible to a wider range of people in the form of web-based libraries. Table 10-1 offers a brief overview of the different libraries in the order in which they were first released.

Table 10-1. Historical overview of different JavaScript-based deep learning libraries (data captured as of August 2019)

	Active years	★ on GitHub	Known for
brain.js	2015–present	9,856	Neural networks, RNNs, LSTMs, and GRUs
ConvNetJS	2014–2016	9,735	Neural networks, CNNs
Synaptic	2014–present	6,571	Neural networks, LSTMs
MXNetJS	2015–2017	420	Running MXNet models
Keras.js	2016–2017	4,562	Running Keras models
CaffeJS	2016–2017	115	Running Caffe models
TensorFlow.js (formerly known as deeplearn.js)	2017–present	11,282	Running TensorFlow models on GPU
ml5.js	2017–present	2,818	Easy to use on top of TF.js.
ONNX.js	2018–present	853	Speed, running ONNX models

Let’s go through a few of these libraries in more detail and see how they evolved.

JavaScript-Based Machine Learning Libraries: A Brief History

Since the breakthrough of deep learning in recent years, many attempts have been made to make AI accessible to a wider range of people in the form of web-based libraries. Table 10-1 offers a brief overview of the different libraries in the order in which they were first released.

Table 10-1. Historical overview of different JavaScript-based deep learning libraries (data captured as of August 2019)

	Active years	★ on GitHub	Known for
brain.js	2015–present	9,856	Neural networks, RNNs, LSTMs, and GRUs
ConvNetJS	2014–2016	9,735	Neural networks, CNNs
Synaptic	2014–present	6,571	Neural networks, LSTMs
MXNetJS	2015–2017	420	Running MXNet models
Keras.js	2016–2017	4,562	Running Keras models
CaffeJS	2016–2017	115	Running Caffe models
TensorFlow.js (formerly known as deeplearn.js)	2017–present	11,282	Running TensorFlow models on GPU
ml5.js	2017–present	2,818	Easy to use on top of TF.js.
ONNX.js	2018–present	853	Speed, running ONNX models

Let’s go through a few of these libraries in more detail and see how they evolved.

ConvNetJS

ConvNetJS is a JavaScript library that was designed in 2014 by Andrej Karpathy as part of a course during his Ph.D. at Stanford University. It trained CNNs in the browser, an exciting proposition, especially in 2014, considering the AI hype was starting to take off, and a developer wouldn't have had to go through an elaborate and painful setup process to get running. ConvNetJS helped introduce AI to so many people for the first time with interactive training demonstrations in the browser.

NOTE

In fact, when MIT scientist Lex Fridman taught his popular self-driving course in 2017, he challenged students worldwide to train a simulated autonomous car using reinforcement learning—in the browser using ConvNetJS—as shown in Figure 10-2.

Figure 10-2. Screenshot of DeepTraffic training a car with reinforcement learning using ConvNetJS

Figure 10-2. Screenshot of DeepTraffic training a car with reinforcement learning using ConvNetJS

Keras.js

Keras.js was introduced in 2016 by Leon Chen. It was a Keras port made to work in the browser by using JavaScript. Keras.js used WebGL to run computations on the GPU. It used shaders (special operations for pixel rendering) to run inferences, which made them run much faster than using just the CPU. Additionally, Keras.js could run on a Node.js server on a CPU to provide server-based inferences. Keras.js implemented a handful of convolutional, dense, pooling, activation, and RNN layers. It is no longer under active development.

Keras.js

Keras.js was introduced in 2016 by Leon Chen. It was a Keras port made to work in the browser by using JavaScript. Keras.js used WebGL to run computations on the GPU. It used shaders (special operations for pixel rendering) to run inferences, which made them run much faster than using just the CPU. Additionally, Keras.js could run on a Node.js server on a CPU to provide server-based inferences. Keras.js implemented a handful of convolutional, dense, pooling, activation, and RNN layers. It is no longer under active development.

ONNX.js

Created by Microsoft in 2018, ONNX.js is a JavaScript library for running ONNX models in browsers and on Node.js. ONNX is an open standard for representing machine learning models that is a collaboration between Microsoft, Facebook, Amazon, and others. ONNX.js is surprisingly fast. In fact, faster than even TensorFlow.js (discussed in the next section) in early benchmarks, as shown in Figure 10-3 and Figure 10-4. This could be attributed to the following reasons:

- ONNX.js utilizes WebAssembly (from Mozilla) for execution on the CPU and WebGL on the GPU.
- WebAssembly allows it to run C/C++ and Rust programs in the web browser while providing near-native performance.
- WebGL provides GPU-accelerated computations like image processing within the browser.
- Although browsers tend to be single-threaded, ONNX.js uses Web Workers to provide a multithreaded environment in the background for parallelizing data operations.

Figure 10-3. Benchmarking data for ResNet-50 on different JavaScript machine learning libraries on CPU (data source)

Figure 10-4. Benchmarking data for ResNet-50 on different JavaScript machine learning libraries on GPU (data source)

I

Figure 10-4. Benchmarking data for ResNet-50 on different JavaScript machine learning libraries on GPU (data source)

TensorFlow.js

Some libraries offered the ability to train within the browser (e.g., ConvNetJS), whereas other libraries offered blazing-fast performance (e.g., the now-defunct TensorFire). deeplearn.js from Google was the first library that supported fast GPU accelerated operations using WebGL while also providing the ability to define, train, and infer within the browser. It offered both an immediate execution model (for inference) as well as a delayed execution model for training (like in TensorFlow 1.x). Originally released in 2017, this project became the core of TensorFlow.js (released in 2018). It is considered an integral part of the TensorFlow ecosystem, and as a result, it is currently the most actively developed JavaScript deep learning library. Considering this fact, we focus on TensorFlow.js in this chapter. To make TensorFlow.js even simpler to use, we also look at ml5.js, which is built on top of TensorFlow.js and abstracts away its complexities, exposing a simple API with ready-to-use models from GANs to PoseNet.

FROM THE CREATOR'S DESK

By Shanqing Cai, senior software engineer at Google and author of *Deep Learning with JavaScript* (Manning)

The forebearer of TensorFlow.js, deeplearn.js, originated from an effort at Google to create an intuitive and interactive visualization to teach people how neural networks are trained. This visualization, today known as "TensorFlow Playground" and available at *https://playground.tensorflow.org*, used an early version of deeplearn.js to train a multilayered neural network entirely in the browser. In building TensorFlow Playground, the engineers became impressed by the potential of using WebGL to perform accelerated training and inference of deep learning models in the browser and at the client side. A team of engineers was assembled at Google to realize this vision, which gave birth to today's TensorFlow.js, a full-blown deep learning library that supports hundreds of operations and dozens of neural network layers and runs on environments ranging from the browser to Node.js, from native-mobile apps to cross-platform desktop apps.

FROM THE CREATOR'S DESK

By Daniel Smilkov, software engineer at Google Brain and coauthor of TensorFlow.js

Before TensorFlow.js, Nikhil [Thorat] and I were building neural network interpretability tools in the browser. To enable a truly interactive experience, we wanted to run inference and compute gradients directly in the browser without sending data to a server. This led to deeplearn.js (you can still see the package with its API on npm), which we released in August 2017. The project got great momentum with creative coders being one of the earliest adopters. Riding on this momentum, the team grew quickly, and six months later we launched TensorFlow.js.

FROM THE CREATOR'S DESK

By Daniel Smilkov, software engineer at Google Brain and coauthor of TensorFlow.js

Before TensorFlow.js, Nikhil [Thorat] and I were building neural network interpretability tools in the browser. To enable a truly interactive experience, we wanted to run inference and compute gradients directly in the browser without sending data to a server. This led to deeplearn.js (you can still see the package with its API on npm), which we released in August 2017. The project got great momentum with creative coders being one of the earliest adopters. Riding on this momentum, the team grew quickly, and six months later we launched TensorFlow.js.

TensorFlow.js Architecture

First, let's take a look at the high-level architecture of TensorFlow.js (see Figure 10-5). TensorFlow.js runs directly in the browser on desktop and mobile. It utilizes WebGL for GPU acceleration, but also can fall back to the browser's runtime for execution on the CPU.

It consists of two APIs: the Operations API and the Layers API. The Operations API provides access to lower-level operations such as tensor arithmetic and othe mathematical operations. The Layers API builds on top of the Operations API to provide layers such as convolution, ReLU, and so on.

Figure 10-5. A high-level overview of the TensorFlow.js and ml5.js ecosystem

Beyond the browser, TensorFlow.js can also run on a Node.js server.

Additionally, ml5.js uses TensorFlow.js to provide an even higher-level API along with several prebuilt models. Having access to all of these APIs at different levels of abstraction allows us to build web apps, not only to do simple inference, but also to train models within the browser itself.

Following are some common questions that come up during the development life cycle for browser-based AI:

- How do I run pretrained models in the browser? Can I use my webcam feed for real-time interactivity?
- How can I create models for the browser from my TensorFlow trained models?
- Can I even train a model in the browser?
- How do different hardware and browsers affect performance?

We answer each of these questions in this chapter, starting with TensorFlow.js before moving on to ml5.js. We explore some rich built-in functionality contributed by the ml5.js community, which would otherwise take a lot of effort and expertise to implement directly on TensorFlow.js. We also look at approaches to benchmarking before looking at some motivating examples built by creative developers.

Now let’s take a look at how to take advantage of pretrained models to make inferences within the browser.

Additionally, ml5.js uses TensorFlow.js to provide an even higher-level API along with several prebuilt models. Having access to all of these APIs at different levels of abstraction allows us to build web apps, not only to do simple inference, but also to train models within the browser itself.

Following are some common questions that come up during the development life cycle for browser-based AI:

- How do I run pretrained models in the browser? Can I use my webcam feed for real-time interactivity?
- How can I create models for the browser from my TensorFlow trained models?
- Can I even train a model in the browser?
- How do different hardware and browsers affect performance?

We answer each of these questions in this chapter, starting with TensorFlow.js before moving on to ml5.js. We explore some rich built-in functionality contributed by the ml5.js community, which would otherwise take a lot of effort and expertise to implement directly on TensorFlow.js. We also look at approaches to benchmarking before looking at some motivating examples built by creative developers.

Now let’s take a look at how to take advantage of pretrained models to make inferences within the browser.

Running Pretrained Models Using TensorFlow.js

TensorFlow.js offers lots of pretrained models that we can directly run in the browser. Some examples include MobileNet, SSD, and PoseNet. In the following example, we load a pretrained MobileNet model. The full code is located on the book's GitHub repository (see *http://PracticalDeepLearning.ai*) a *code/chapter-10/mobilenet-example/*.

First, we import the latest bundle of the library:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/
tfjs@latest/dist/tf.min.js"></script>
```

We then import the MobileNet model:

```
<script src="https://cdn.jsdelivr.net/npm/
@tensorflow-models/mobilenet@1.0.0"></script>
```

Now we can make predictions using the following code:

```
<img id="image" src="cat.jpg" />
<p id="prediction_output">Loading predictions...</p>
<script>
 const image = document.getElementById("image");
 const predictionOutput = document.getElementById("prediction_output");

 // Load the model.
 mobilenet.load().then(model => {
   // Classify the image. And output the predictions
   model.classify(image).then(predictions => {
     predictionOutput.innerText = predictions[0].className;
   });
 });
</script>
```

Figure 10-6 shows a sample output.

t

[{"className":"Egyptian cat","probability":0.3447355628013611},
{"className":"cougar, puma, catamount, mountain lion, painter, panther, Felis concolor","probability":0.10490904748439789},
{"className":"Persian cat","probability":0.06154840067028999}]

Figure 10-6. Class prediction output in the browser

We can alternatively load a model using a JSON file URL in the following manner:

```
const path = 'https://storage.googleapis.com/tfjs-
models/tfjs/mobilenet_v1_1.0_224/model.json';

model = tf.loadLayersModel(path).then(model => {
    // Load model and output predictions here
});
```

The JSON file contains the architecture, the parameter names of the model, and the paths to the smaller sharded weight files. The sharding allows the files to be small enough to be cached by web browsers, which would make loading faster for any subsequent time the model would be needed.

Model Conversion for the Browser

In the previous section, we examined how to load a pretrained model that is already in the JSON format. In this section, we learn how to convert a pretrained Keras model (`.h5` format) to JSON that is compatible with TensorFlow.js. To do this, we need to install the conversion tool using `pip`.

```
$ pip install tensorflowjs
```

Assuming that our trained Keras model is stored in a folder named *keras_model*, we would be able to use the following command to convert it:

```
$ tensorflowjs_converter --input_format keras keras_model/model.h5 web_model/
```

Now the `web_model` directory will contain the `.json` and `.shard` files that we can easily load using the `tf.loadLayersModel` method:

```
$ ls web_model
group1-shard1of4  group1-shard3of4  model.json group1-shard2of4  group1-shard4of4
```

That's it! Bringing our trained model to the browser is an easy task. For cases in which we don't already have an existing trained model, TensorFlow.js also allows us to train models directly in the browser. In the next section, we explore this by creating an end-to-end example of training a model using a webcam feed

TIP

Loading a model locally requires running a web server. There are tons of options that we can use, ranging from the LAMP (Linux, Apache, MySQL, PHP) stack to installing `http-server` using npm, to even running Internet Information Services (IIS) on Windows to test model loading locally. Even Python 3 can run a simple web server:

```
$ python3 -m http.server 8080
```

Training in the Browser

The previous example used a pretrained model. Let's take it up a notch and train our own models directly in the browser using input from the webcam. Like in some of the previous chapters, we look at a simple example in which we exploit transfer learning to make the training process faster.

Adapted from Google's Teachable Machine, we use transfer learning to construc a simple binary classification model, which will be trained using the webcam feed. To build this, we need a feature extractor (converts input images into features or embeddings), and then attach a network that converts these features into a prediction. Finally, we can train it with webcam inputs. The code is referenced in the book's GitHub repository (see *http://PracticalDeepLearning.ai*) at *code/chapter-10/teachable-machine*.

NOTE

Google Creative Lab built a fun interactive website called Teachable Machine where the user can train three classes on any type of classification problem by simply showing those objects in front of the webcam. The three classes are given simple labels of green, purple, and orange. During prediction, rather than showing bland-looking class probabilities in text on a web page (or even worse, in the console), Teachable Machine shows GIFs of cute animals or plays different sounds based on the class that is being predicted. As you can imagine, this would be a fun and engaging experience for kids in a classroom, and it would serve as a wonderful tool to introduce them to AI.

Figure 10-7. Training live in the browser with Teachable Machine

Figure 10-7. Training live in the browser with Teachable Machine

Feature Extraction

As we explored in some of the early chapters in this book, training a large model from scratch is a slow process. It's a lot cheaper and quicker to use a pretrained model and use transfer learning to customize the model for our use case. We'll use that model to extract high-level features (embeddings) from the input image, and use those features to train our custom model.

For extracting the features, we load and use a pretrained MobileNet model:

```
const path = 'https://storage.googleapis.com/tfjs-
models/tfjs/mobilenet_v1_1.0_224/model.json';
const mobilenet = await tf.loadLayersModel(path);
```

Let's inspect the inputs and outputs of the model. We know that the model was trained on ImageNet and the final layer predicts a probability of each one of the 1,000 classes:

```
const inputLayerShape = mobilenet.inputs[0].shape; // [null, 224, 224, 3]
const outputLayerShape = mobilenet.outputs[0].shape; // [null, 1000]
const numLayers = mobilenet.layers.length; // 88
```

To extract the features, we select a layer closer to the output. Here we select `conv_pw_13_relu` and make it the output of the model; that is, remove the dense layers at the end. The model we created is called a feature extraction model:

```
// Get a specific layer
const layer = mobilenet.getLayer('conv_pw_13_relu');

// Create a new feature extraction model
featureExtractionModel = tf.model({inputs: mobilenet.inputs, outputs:
layer.output});

featureExtractionModel.layers.length; // 82
```

We'll keep the feature extraction model unmodified during our training process. Instead, we add a trainable set of layers on top of it to build our classifier:

```
const trainableModel = tf.sequential({
    layers: [
```

```
            tf.layers.flatten({inputShape: [7, 7, 1024]}),
            tf.layers.dense({
            units: 64,
            activation: 'relu',
            kernelInitializer: 'varianceScaling',
            useBias: true
        }),
        tf.layers.dense({
            units: 2,
            kernelInitializer: 'varianceScaling',
            useBias: false,
            activation: 'softmax'
        })]
    });
```

```
        tf.layers.flatten({inputShape: [7, 7, 1024]}),
        tf.layers.dense({
        units: 64,
        activation: 'relu',
        kernelInitializer: 'varianceScaling',
        useBias: true
    }),
    tf.layers.dense({
        units: 2,
        kernelInitializer: 'varianceScaling',
        useBias: false,
        activation: 'softmax'
    })]
});
```

Data Collection

Here, we collect images using the webcam feed and process them for feature extraction. The `capture()` function in the Teachable Machine is responsible for setting up the `webcamImage` for storing the captured images from the webcam in memory. Now, let's preprocess them to make them applicable to the feature extraction model:

```
function capture() {
     return tf.tidy(() => {
         // convert to a tensor
         const webcamImage = tf.fromPixels(webcam);
         // crop to 224x224
         const croppedImage = cropImage(webcamImage);
         // create batch and normalize
         const batchedImage = croppedImage.expandDims(0);
         return batchedImage.toFloat().div(tf.scalar(127)).sub(tf.scalar(1));
     });
}
```

After we capture the images, we can add the image and label to the training data

```
function addTrainingExample(img, label) {
     // Extract features.
     const data = featureExtractionModel.predict(img);
     // One-hot encode the label.
     const oneHotLabel = tf.tidy(() =>
     tf.oneHot(tf.tensor1d([label], 'int32'), 2));
     // Add the label and data to the training set.
}
```

Data Collection

Here, we collect images using the webcam feed and process them for feature extraction. The `capture()` function in the Teachable Machine is responsible for setting up the `webcamImage` for storing the captured images from the webcam in memory. Now, let's preprocess them to make them applicable to the feature extraction model:

```
function capture() {
     return tf.tidy(() => {
         // convert to a tensor
         const webcamImage = tf.fromPixels(webcam);
         // crop to 224x224
         const croppedImage = cropImage(webcamImage);
         // create batch and normalize
         const batchedImage = croppedImage.expandDims(0);
         return batchedImage.toFloat().div(tf.scalar(127)).sub(tf.scalar(1));
     });
}
```

After we capture the images, we can add the image and label to the training data:

```
function addTrainingExample(img, label) {
     // Extract features.
     const data = featureExtractionModel.predict(img);
     // One-hot encode the label.
     const oneHotLabel = tf.tidy(() =>
     tf.oneHot(tf.tensor1d([label], 'int32'), 2));
     // Add the label and data to the training set.
}
```

Training

Next, we train the model, much as we trained in Chapter 3. Just like in Keras and TensorFlow, we add an optimizer and define the loss function:

```
const optimizer = tf.train.adam(learningRate);
model.compile({ optimizer: optimizer, loss: 'categoricalCrossentropy' });
model.fit(data, label, {
    batchSize,
    epochs: 5,
    callbacks: {
        onBatchEnd: async (batch, logs) => {
            await tf.nextFrame();
        }
    }
}
```

NOTE

One important thing to keep in mind is that on the GPU, memory allocated by TensorFlow.js is not released when a `tf.tensor` object goes out of scope. One solution is to call the `dispose()` method on every single object created. However, that would make the code more difficult to read, particularly when chaining is involved. Consider the following example:

```
const result = a.add(b).square().neg();
return result;
```

To cleanly dispose of all memory, we'd need to break it down into the following:

```
const sum = a.add(b);
const square = sum.square();
const result = square.neg();
sum.dispose();
square.dispose();
return result;
```

Instead, we could simply use `tf.tidy()` to do the memory management for us, while keeping our code clean and readable. Our first line simply needs to be wrapped inside the `tf.tidy()` block, as shown here:

```
const result = tf.tidy(() => {
    return a.add(b).square().neg();
});
```

With a CPU backend, the objects are automatically garbage collected by the browser. Calling `.dispose()` there does not have any effect.

As a simple use case, let's train the model to detect emotions. To do so, we need to simply add training examples belonging to either one of two classes: happy or sad. Using this data, we can start training. Figure 10-8 shows the final result after training the model on 30 images per each class.

Figure 10-8. Predictions by our model on a webcam feed within a browser

TIP

Usually, when using a webcam for predictions, the UI tends to freeze. This is because the computation happens on the same thread as the UI rendering. Calling `await tf.nextFrame()` will release the UI thread, which will make the web page responsive and prevent the tab/browser from freezing.

With a CPU backend, the objects are automatically garbage collected by the browser. Calling `.dispose()` there does not have any effect.

As a simple use case, let's train the model to detect emotions. To do so, we need to simply add training examples belonging to either one of two classes: happy or sad. Using this data, we can start training. Figure 10-8 shows the final result after training the model on 30 images per each class.

Figure 10-8. Predictions by our model on a webcam feed within a browser

TIP

Usually, when using a webcam for predictions, the UI tends to freeze. This is because the computation happens on the same thread as the UI rendering. Calling `await tf.nextFrame()` will release the UI thread, which will make the web page responsive and prevent the tab/browser from freezing.

GPU Utilization

We can see the CPU/GPU utilization during training and inference using the Chrome profiler. In the previous example, we recorded the utilization for 30 seconds and observed the GPU usage. In Figure 10-9, we see that the GPU is used a quarter of the time.

Figure 10-9. GPU utilization shown in the Google Chrome profiler view

So far, we've discussed how to do everything from scratch, including loading the model, capturing video from the webcam, collecting the training data, training the model, and running an inference. Wouldn't it be great if all these steps could

be taken care of under the hood and we could just focus on what to do with the results of the inference? In the next section, we discuss exactly that with ml5.js.

e

be taken care of under the hood and we could just focus on what to do with the results of the inference? In the next section, we discuss exactly that with ml5.js.

ml5.js

ml5.js is a higher abstraction of TensorFlow.js that makes it easy to use existing pretrained deep learning models in a unified way, with a minimal number of lines of code. The package comes with a wide range of built-in models, ranging from image segmentation to sound classification to text generation, as shown in Table 10-2. Further, ml5.js reduces the steps related to preprocessing, postprocessing, and so on, letting us concentrate on building the application that we want with these models. For each of these functionalities, ml5js comes with a demonstration and reference code.

Table 10-2. Selected built-in models in ml5.js, showing the range of functionalities in text, image, and sound

Functionality	Description
PoseNet	Detect the location of human joints
U-Net	Object segmentation; e.g., removing object background
Style Transfer	Transfers style of one image to another
Pix2Pix	Image-to-image translation; e.g., black-and-white to color
Sketch RNN	Creates doodles based on an incomplete sketch
YOLO	Object detection; e.g., locates faces with bounding boxes
Sound Classifier	Recognizes audio; e.g., whistle, clap, “one,” “stop,” etc.
Pitch Detector	Estimates pitch of sound
Char RNN	Generates new text based on training on a large corpus of text
Sentiment Classifier	Detects sentiment of a sentence
Word2Vec	Produces word embeddings to identify word relations
Feature Extractor	Generates features or embeddings from input
kNN Classifier	Creates a fast classifier using *k*-Nearest Neighbor

Let’s see it in action. First, we import the latest bundle of ml5.js, which is simila to TensorFlow.js:

```
<script src="https://unpkg.com/ml5@latest/dist/ml5.min.js"
type="text/javascript"></script>
```

Note that we no longer need to import anything related to TensorFlow.js, because it already comes included with ml5.js. We create a simple example where we use the same MobileNet scenario as earlier:

```
// Initialize the image classifier method with MobileNet
const classifier = ml5.imageClassifier('MobileNet', modelLoaded);

// Make a prediction with the selected image
classifier.predict(document.getElementById('image'), function(err, results) {
  console.log(results);
});
```

Done! In effectively three lines, a pretrained model is running in our browser. Now, let's open the browser's console to inspect the output presented in Figure 10-10.

Figure 10-10. The top predicted classes with the probability of each class

TIP

If you are unfamiliar with the browser console, you can simply access it by right-clicking anywhere within the browser window and selecting "Inspect element." A separate window opens with the console inside it.

We can find the full source code for the previous example at *code/chapter-10/ml5js*.

Note that ml5.js uses callbacks to manage asynchronous calls of the models. A callback is a function that is executed after the accompanying call is finished. For instance, in the last code snippet, after the model is loaded, the `modelLoaded` function is called, indicating that the model is loaded into memory.

NOTE

p5.js is a library that works nicely in conjunction with ml5.js and makes it super easy to make model predictions in real time using a live video stream. You can find a code snippet demonstrating the power of p5.js at *code/chapter-10/p5js-webcam/*.

ml5.js natively supports p5.js elements and objects. You can use p5.js elements for drawing objects, capturing webcam feeds, and more. Then, you can easily use such elements as input to the ml5.js callback functions.

ml5.js natively supports p5.js elements and objects. You can use p5.js elements for drawing objects, capturing webcam feeds, and more. Then, you can easily use such elements as input to the ml5.js callback functions.

PoseNet

So far in this book, we have primarily explored image classification problems. In later chapters, we take a look at object detection and segmentation problems. These types of problems form a majority of computer-vision literature. In this section, however, we choose to take a break from the usual and deal with a different kind of problem: keypoint detection. This has significant applications in a variety of areas including health care, fitness, security, gaming, augmented reality, and robotics. For instance, to encourage a healthy lifestyle through exercise, Mexico City installed kiosks that detect the squat pose and offer free subway tickets to passengers who can do at least 10 squats. In this section, we explore how to run something so powerful in our humble web browser.

The PoseNet model offers real-time pose estimation in the browser. A “pose” consists of the position of different keypoints (including joints) in the human body such as the top of the head, eyes, nose, neck, wrists, elbows, knees, ankles, shoulders, and hips. You can use PoseNet for single or multiple poses that might exist in the same frame.

Let’s build an example using PoseNet, which is readily available in ml5.js, to detect and draw keypoints (with the help of p5.js).

Figure 10-11. Keypoints drawn using PoseNet on a picture of former President Obama in a snowball fight

You can find the code to detect keypoints for a still image at *code/chapter-10/posenet/single.html*:

```
<script src="http://p5js.org/assets/js/p5.min.js"></script>
```

```
<script src="http://p5js.org/assets/js/p5.dom.min.js"></script>
<script src="https://unpkg.com/ml5@latest/dist/ml5.min.js"></script>

<script>
function setup() {
    // Set up camera here

    // Call PoseNet model
    const poseNet = ml5.poseNet(video, modelReady);

    // PoseNet callback function
    poseNet.on('pose', function (results) {
        const poses = results;
    });
}
</script>
```

We can also run a similar script (at *code/chapter-10/posenet/webcam.html*) on our webcam.

Now let us look at another example that is supported by ml5.js.

FROM THE CREATOR'S DESK

By Cristobal Valenzuela, cofounder of Runway and contributor to ml5.js

ml5.js started as an experiment within NYU's Interactive Telecommunications Program (ITP) to understand creative applications of machine learning in the browser. Heavily inspired by Processing and p5.js, the original idea was to create a series of code experiments with deeplearn.js (now TensorFlow.js) and p5.js. In August of 2017, a few days after deeplearn.js was released, I created a repository called "p5deeplearn: deeplearn.js meets p5" as a tentative first approach. This repository was the genesis of ml5.js, with the "5" paying homage to p5.js and the Processing philosophy itself.

With the guidance of Dan Shiffman and the support of a Google Faculty Research Award to collaborate with the TensorFlow.js team, we organized a weekly working group to discuss developing the library itself and invited guest artists and researchers to participate in its development. The main idea behind ml5.js has always been to create an approachable and easy-to-use library space and community where we could encourage experimentation

and creation with AI in the browser.

FROM THE CREATOR'S DESK

By Daniel Shiffman, associate arts professor at the Interactive Telecommunications Program at NYU's Tisch School of the Arts and a director of The Processing Foundation

Despite the ease of running a machine learning model in the browser with TensorFlow.js, beginners who want to code their own still face the challenge of mathematical notation and low-level technical code. Machine learning frameworks are commonly geared toward people with knowledge of calculus, linear algebra, statistics, data science, and several years of programming in a language like Python or C++. Although important for research and development of new machine learning models and architectures, starting from this point can turn away newcomers with other backgrounds. Rather than thinking creatively about how to use machine learning as an artistic platform, beginners can be overwhelmed by fine distinctions between scalars, vectors, matrices, operations, inputs layers, outputs, and more.

This is where ml5.js comes in. Its goal is to provide a neighborly approach to creating and exploring AI in the browser. *The "5" in ml5 is an homage to p5.js, a JavaScript library that serves as the primary inspiration and model for ml5.* p5.js is maintained by the Processing Foundation whose mission is "to empower people of all interests and backgrounds to learn how to program and make creative work with code, especially those who might not otherwise have access to these tools and resources."

With Processing, an artist can invent their own tools rather than relying on those created and maintained by others. Today, being "software literate" is arguably more important than ever. Algorithms affect our lives in ways we couldn't have anticipated, and the explosion of machine learning research has further extended their reach. Our interactions are now negotiated not just with each other, but with autonomous vehicles, ever-listening assistants, and cameras that identify our faces and poses. Without access to and understanding of the machine learning models, underlying data, and outputs

driving the software, how can we meaningfully engage, question, and propose alternatives? Machine learning faces a similar challenge of approachability as simply learning to code did more than 15 years ago.

The development of ml5.js is funded by a Google Faculty Research Award at ITP. ITP is a two-year graduate program in the NYU Tisch School of the Arts, whose mission is to explore the imaginative use of communications technologies—how they might augment, improve, and bring delight and art into people's lives. Each week, a group of 10 to 15 ITP students, along with outside contributors and guest artists, meets at ITP to discuss API decisions, share creative projects, and teach each other about contributing to open source. To date, there have been more than 50 contributors to 17 releases of ml5.js.

driving the software, how can we meaningfully engage, question, and propose alternatives? Machine learning faces a similar challenge of approachability as simply learning to code did more than 15 years ago.

The development of ml5.js is funded by a Google Faculty Research Award at ITP. ITP is a two-year graduate program in the NYU Tisch School of the Arts, whose mission is to explore the imaginative use of communications technologies—how they might augment, improve, and bring delight and art into people's lives. Each week, a group of 10 to 15 ITP students, along with outside contributors and guest artists, meets at ITP to discuss API decisions, share creative projects, and teach each other about contributing to open source. To date, there have been more than 50 contributors to 17 releases of ml5.js.

pix2pix

“Hasta la vista, baby!”

This is one of the most memorable lines in the history of film. Coincidentally, it was uttered by an AI cyborg in the 1991 classic *Terminator 2: Judgment Day*. By the way, its translation is “Goodbye, baby!” Language translation technology has come a long way since then. It used to be that language translation was built on phrase substitution rules. And now, it’s replaced by much better performing deep learning systems that understand the context of the sentence to convert it into a similar meaning sentence in the target language.

Here’s a thought: if we can translate from sentence one to sentence two, could we translate a picture from one setting to another? Could we do the following:

- Convert an image from low resolution to higher resolution?
- Convert an image from black and white to color?
- Convert an image from daytime to nighttime view?
- Convert a satellite image of the earth into a map view?
- Convert an image from a hand-drawn sketch into a photograph?

Well, image translation is not science fiction anymore. In 2017, Philip Isola et al developed a way to convert a picture into another picture, conveniently naming it pix2pix. By learning from several pairs of before and after pictures, the pix2pix model is able to generate highly realistic renderings based on the input image. For example, as demonstrated in Figure 10-12, given a pencil sketch of a bag, it can recreate the photo of a bag. Additional applications include image segmentation, synthesizing artistic imagery, and more.

Figure 10-12. Example of input and output pairs on pix2pix

NOTE

Imagine a scenario with a bank teller and a currency counterfeiter. The job of the bank teller is to spot fake currency bills, whereas the counterfeiter's goal is to make it as difficult as possible for the bank teller to identify the fakes. They are clearly in an adversarial situation. Each time the cop spots the fake bills, the counterfeiter learns his mistake, takes it as an opportunity to improve (growth mindset after all), and tries to make it even more difficult for the bank teller to thwart him next time. This forces the bank teller to get better at recognizing fakes over time. This feedback cycle forces both of them to get better at what they do. This is the underlying principle driving GANs.

As Figure 10-13 illustrates, GANs consist of two networks, a Generator and a Discriminator, which have the same adversarial relationship as the counterfeiter and the bank teller. The Generator's job is to generate realistic-looking output, very similar to the training data. The Discriminator's responsibility is to identify whether the data passed to it by the Generator was real or fake. The output of the Discriminator is fed back into the Generator to begin the next cycle. Each time the Discriminator correctly identifies a generated output as a fake, it forces the Generator to get better in the next cycle.

It is worthwhile to note that GANs typically do not have control over the data to be generated. However, there are variants of GANs, such as *conditional GANs*, that allow for labels to be part of the input, providing more control over the output generation; that is, conditioning the output. pix2pix is an example of a conditional GAN.

Figure 10-13. A flowchart for a GAN

We used pix2pix to create a simple sketching app that works in the browser. The output images are really interesting to look at. Consider the examples shown in Figure 10-14 and Figure 10-15.

Figure 10-14. Sketch-to-image example

Figure 10-15. We can create colored blueprints (left) and pix2pix will convert them to realistic-looking human faces (right)

NOTE

Fun fact: Ian Goodfellow came up with the idea for GANs while at a bar. This adds yet another item to the list of inventions, organizations, and companies whose ideas originated over drinks, including the creation of the RSA Algorithm, Southwest Airlines, and the game of Quidditch.

pix2pix works by training on pairs of images. In Figure 10-16, the image on the left is the input image or the conditional input. The image on the right is the target image, the realistic output that we want to generate (if you are reading the print version, you will not see the color image on the right).

Figure 10-16. Training pairs for pix2pix: a B&W image and its original color image

One of the easier ports for training pix2pix is the TensorFlow-based implementation by Christopher Hesse. We can use a very simple script to train our own model:

```
python pix2pix.py \
  --mode train \
  --output_dir facades_train \
  --max_epochs 200 \
  --input_dir facades/train \
  --which_direction BtoA
```

After training has finished, we can save the model using the following command:

```
python tools/export-checkpoint.py --checkpoint ../export --output_file
models/MY_MODEL_BtoA.pict
```

After that, we can use this simple code to load the saved weights to ml5.js. Note the transfer function that is used to retrieve the output in a canvas:

```
// Create a pix2pix model using a pre-trained network
const pix2pix = ml5.pix2pix('models/customModel.pict', modelLoaded);

// Transfer using a canvas
pix2pix.transfer(canvas, function(err, result) {
  console.log(result);
```

```
});
```

We can also draw strokes and allow real-time sketching. For instance, Figure 10-17 shows an example that draws Pikachu.

Figure 10-17. Pix2Pix: Edges to Pikachu by Yining Shi, built on ml5.js

FROM THE CREATOR'S DESK

By Cristobal Valenzuela, cofounder of Runway and contributor to ml5.js

There have been a lot of interesting projects, experiments, and creations built with ml5.js: artists have been using it to create art as well as live stage performances. Nikita Huggins and Ayodamola Okunseinde, two New York-based artists, have been exploring ways of increasing the diversity of the models the library currently supports, so students and developers have built applications for Hackathons and for all kinds of competitions. But what's super exciting is to see how it has been used in curricula around the world to teach about machine learning in an approachable manner, opening the possibilities for a broad audience of artists, creative coders, and students.

```
});
```

We can also draw strokes and allow real-time sketching. For instance, Figure 10-17 shows an example that draws Pikachu.

Figure 10-17. Pix2Pix: Edges to Pikachu by Yining Shi, built on ml5.js

FROM THE CREATOR'S DESK

By Cristobal Valenzuela, cofounder of Runway and contributor to ml5.js

There have been a lot of interesting projects, experiments, and creations built with ml5.js: artists have been using it to create art as well as live stage performances. Nikita Huggins and Ayodamola Okunseinde, two New York-based artists, have been exploring ways of increasing the diversity of the models the library currently supports, so students and developers have built applications for Hackathons and for all kinds of competitions. But what's super exciting is to see how it has been used in curricula around the world to teach about machine learning in an approachable manner, opening the possibilities for a broad audience of artists, creative coders, and students.

Benchmarking and Practical Considerations

As people who care deeply about how our end users perceive our product, it's important for us to treat them right. Two factors play a large role in how users experience our product: the model size, and the inference time based on the hardware. Let's take a closer look at each factor.

Benchmarking and Practical Considerations

As people who care deeply about how our end users perceive our product, it's important for us to treat them right. Two factors play a large role in how users experience our product: the model size, and the inference time based on the hardware. Let's take a closer look at each factor.

Model Size

A typical MobileNet model is 16 MB. Loading this on a standard home or office network might just take a few seconds. Loading the same model on a mobile network would take even longer. The clock is ticking, and the user is becoming impatient. And this is before the model even gets a chance to start inference. Waiting for big models to load is more detrimental to the UX than their runtime, especially where internet speeds are not as fast as a broadband paradise like Singapore. There are a few strategies that can help:

Pick the smallest model for the job

Among pretrained networks, EfficientNet, MobileNet, or SqueezeNet tend to be the smallest (in order of decreasing accuracy).

Quantize the model

Reduce the model size using the TensorFlow Model Optimization Toolkit before exporting to TensorFlow.js.

Build our own tiny model architecture

If the final product does not need heavy ImageNet-level classification, we could build our own smaller model. When Google made the J.S. Bach doodle on the Google home page, its model was only 400 KB, loading almost instantly.

Model Size

A typical MobileNet model is 16 MB. Loading this on a standard home or office network might just take a few seconds. Loading the same model on a mobile network would take even longer. The clock is ticking, and the user is becoming impatient. And this is before the model even gets a chance to start inference. Waiting for big models to load is more detrimental to the UX than their runtime, especially where internet speeds are not as fast as a broadband paradise like Singapore. There are a few strategies that can help:

Pick the smallest model for the job

Among pretrained networks, EfficientNet, MobileNet, or SqueezeNet tend to be the smallest (in order of decreasing accuracy).

Quantize the model

Reduce the model size using the TensorFlow Model Optimization Toolkit before exporting to TensorFlow.js.

Build our own tiny model architecture

If the final product does not need heavy ImageNet-level classification, we could build our own smaller model. When Google made the J.S. Bach doodle on the Google home page, its model was only 400 KB, loading almost instantly.

Inference Time

Considering our model is accessible in a browser running on a PC or a mobile phone, we would want to pay careful attention to the UX, especially on the slowest hardware. During our benchmarking process, we ran *chapter10/code/benchmark.html* within various browsers on different devices. Figure 10-18 presents the results of these experiments.

Figure 10-18. Inference time for MobileNetV1 in Chrome on different devices

Figure 10-18 implies the faster the hardware, the faster the model inference. Apple appears to be outdoing Android in terms of GPU performance. Though, clearly, it is not an "apples-to-apples comparison."

Out of curiosity, do different browsers run inference at the same speed? Let's find that out on an iPhone X; Figure 10-19 shows the results.

Figure 10-19. Inference time in different browsers on iPhone X

Figure 10-19 shows us the same speed in all browsers on an iPhone. This shouldn't be surprising, because all of these browsers use iPhone's WebKit-based built-in browser control called `WKWebView`. How about on a MacBook Pro? Take a look at Figure 10-20 to see.

Figure 10-20. Inference time in different browsers on an i7 @ 2.6 GHz macOS 10.14 machine

The results might be surprising. Chrome is almost double the speed of Firefox in this example. Why is that? Opening a GPU monitor showed that Chrome had much higher GPU utilization compared to Firefox and slightly higher than Safari. The higher the utilization, the faster the inference. What this means is tha depending on the operating system, browsers might have different optimizations to speed up the inference on the GPU, leading to different running times.

One key point to note is that these tests were performed on top-of-the-line devices. They do not necessarily reflect the kind of device an average user might have. This also has implications for battery usage, if run for prolonged periods o time. Accordingly, we need to set appropriate expectations regarding performance, particularly for real-time user experiences.

Figure 10-19. Inference time in different browsers on iPhone X

Figure 10-19 shows us the same speed in all browsers on an iPhone. This shouldn't be surprising, because all of these browsers use iPhone's WebKit-based built-in browser control called `WKWebView`. How about on a MacBook Pro? Take a look at Figure 10-20 to see.

Figure 10-20. Inference time in different browsers on an i7 @ 2.6 GHz macOS 10.14 machine

The results might be surprising. Chrome is almost double the speed of Firefox in this example. Why is that? Opening a GPU monitor showed that Chrome had much higher GPU utilization compared to Firefox and slightly higher than Safari. The higher the utilization, the faster the inference. What this means is that depending on the operating system, browsers might have different optimizations to speed up the inference on the GPU, leading to different running times.

One key point to note is that these tests were performed on top-of-the-line devices. They do not necessarily reflect the kind of device an average user might have. This also has implications for battery usage, if run for prolonged periods of time. Accordingly, we need to set appropriate expectations regarding performance, particularly for real-time user experiences.

Case Studies

Now that we know all the ingredients for deep learning on the browser, let’s see what the industry is cooking.

Case Studies

Now that we know all the ingredients for deep learning on the browser, let’s see what the industry is cooking.

Semi-Conductor

Have you ever dreamed of conducting the New York Philharmonic Orchestra? With Semi-Conductor, your dream is half-way fulfilled. Open the website, stand in front of the webcam, wave your arms, and watch the entire orchestra perform Mozart's Eine Kleine Nachtmusik at your whim! As you might have guessed, it's using PoseNet to track the arm movements and using those movements to set the tempo, volume, and the section of instruments (Figure 10-21) playing the music (including violins, violas, cellos, and double bass). Built by the Google Creative Lab in Sydney, Australia, it uses a prerecorded musical piece broken up into tiny fragments, and each fragment is played at the scored speed and volume depending on the arm movements. Moving hands up increases volume, moving faster increases tempo. This interactive experience is possible only because PoseNet is able to run inferences at several frames a second (on a regular laptop).

Figure 10-21. Control an orchestra by waving your arms on the Semi-Conductor demonstrator

t

)

TensorSpace

CNNs can often feel…well, convoluted. Often treated as a black box, they can be difficult to understand. What do the filters look like? What activates them? Why did they make a certain prediction? They are shrouded in mystery. As with anything complex, visualizations can help open this black box and make it easier to understand. And that's where TensorSpace, the library that "presents tensors in space," comes in.

It allows us to load models in 3D space, explore their structures in the browser, zoom and rotate through them, feed inputs, and understand how the image is processed and passed layer by layer all the way to the final prediction layer. The filters can finally be opened up to manual inspection without the need for any installation. And, as Figure 10-22 teases, if you're feeling savvy, you could even load this in virtual reality against any background!

Figure 10-22. LeNet model visualized inside TensorSpace

TensorSpace

CNNs can often feel…well, convoluted. Often treated as a black box, they can be difficult to understand. What do the filters look like? What activates them? Why did they make a certain prediction? They are shrouded in mystery. As with anything complex, visualizations can help open this black box and make it easier to understand. And that's where TensorSpace, the library that "presents tensors in space," comes in.

It allows us to load models in 3D space, explore their structures in the browser, zoom and rotate through them, feed inputs, and understand how the image is processed and passed layer by layer all the way to the final prediction layer. The filters can finally be opened up to manual inspection without the need for any installation. And, as Figure 10-22 teases, if you're feeling savvy, you could even load this in virtual reality against any background!

Figure 10-22. LeNet model visualized inside TensorSpace

Metacar

Self-driving cars are a complex beast. And using reinforcement learning to train them can take a lot of time, money, and monkeywrenching (not counting the crashes, initially). What if we could train them in the browser itself? Metacar solves this by providing a simulated 2D environment to train toy cars with reinforcement learning, all in the browser, as depicted in Figure 10-23. Just as with video games you progress to ever-more difficult levels, Metacar allows building multiple levels to improve the performance of your car. Utilizing TensorFlow.js, this is aimed at making reinforcement learning more accessible (in which we dive into greater detail in Chapter 17 while building a small-scale autonomous car).

Figure 10-23. Metacar environment for training with reinforcement learning

Airbnb’s Photo Classification

Airbnb, the online property rental company, requires homeowners and renters to upload pictures of themselves for their profiles. Unfortunately, a few people try to find the most readily available picture they have—their driver’s license or passport. Given the confidential nature of the information, Airbnb uses a neural network running on TensorFlow.js to detect sensitive images and prevent their upload to the server.

Airbnb’s Photo Classification

Airbnb, the online property rental company, requires homeowners and renters to upload pictures of themselves for their profiles. Unfortunately, a few people try to find the most readily available picture they have—their driver’s license or passport. Given the confidential nature of the information, Airbnb uses a neural network running on TensorFlow.js to detect sensitive images and prevent their upload to the server.

GAN Lab

Similar to TensorFlow Playground (an in-browser neural network visualization tool), GAN Lab (Figure 10-24) is an elegant visualization tool for understanding GANs using TensorFlow.js. Visualizing GANs is a difficult process, so to simplify it, GAN Lab attempts to learn simple distributions and visualizes the generator and discriminator network output. For instance, the real distribution could be points representing a circle in 2D space. The generator starts from a random Gaussian distribution and gradually tries to generate the original distribution. This project is a collaboration between Georgia Tech and Google Brain/PAIR (People + AI Research).

Figure 10-24. Screenshot of GAN Lab

GAN Lab

Similar to TensorFlow Playground (an in-browser neural network visualization tool), GAN Lab (Figure 10-24) is an elegant visualization tool for understanding GANs using TensorFlow.js. Visualizing GANs is a difficult process, so to simplify it, GAN Lab attempts to learn simple distributions and visualizes the generator and discriminator network output. For instance, the real distribution could be points representing a circle in 2D space. The generator starts from a random Gaussian distribution and gradually tries to generate the original distribution. This project is a collaboration between Georgia Tech and Google Brain/PAIR (People + AI Research).

Figure 10-24. Screenshot of GAN Lab

Summary

We first examined the evolution of JavaScript-based deep learning libraries and chose TensorFlow.js as the candidate to focus on. We ran pretrained models in real time on the webcam feed, and then even trained models in the browser. Tools like Chrome's profiler gave us insight into GPU usage. Then, to simplify development further, we used ml5.js, which enabled us to build demos like PoseNet and pix2pix in just a few lines of code. Finally, we benchmarked the performance of these models and libraries in the real world, ending with some interesting case studies.

One huge benefit of running neural networks in the browser is the vast reach that browsers have compared to any smartphone platform. Add to that the advantage of not having to overcome the user's reluctance to install yet another app. This also makes for a quick prototyping platform that enables inexpensive validation of hypotheses before investing significant amounts of time and money to build native experiences. TensorFlow.js, in conjunction with ml5.js, has accelerated the process of bringing the power of AI to the browser and broaden its reach to the masses.

Summary

We first examined the evolution of JavaScript-based deep learning libraries and chose TensorFlow.js as the candidate to focus on. We ran pretrained models in real time on the webcam feed, and then even trained models in the browser. Tools like Chrome's profiler gave us insight into GPU usage. Then, to simplify development further, we used ml5.js, which enabled us to build demos like PoseNet and pix2pix in just a few lines of code. Finally, we benchmarked the performance of these models and libraries in the real world, ending with some interesting case studies.

One huge benefit of running neural networks in the browser is the vast reach that browsers have compared to any smartphone platform. Add to that the advantage of not having to overcome the user's reluctance to install yet another app. This also makes for a quick prototyping platform that enables inexpensive validation of hypotheses before investing significant amounts of time and money to build native experiences. TensorFlow.js, in conjunction with ml5.js, has accelerated the process of bringing the power of AI to the browser and broaden its reach to the masses.

Chapter 11. Real-Time Object Classification on iOS with Core ML

So far, we have seen our deep learning models running on the desktop, the cloud and the browser. Although there are definite upsides to such a setup, it might not be ideal for all scenarios. In this chapter, we explore making predictions using deep learning models on mobile devices.

Bringing the computation closer to the user's device, rather than a distant remote server, can be advantageous for many reasons:

Latency and interactivity

Sending an image, processing it in the cloud, and returning the result can take several seconds depending on the network quality and quantity of data being transferred. This can make for a poor UX. Decades of UX research, including Jakob Nielsen's findings in 1993, published in his book *Usability Engineering* (Elsevier), showed the following:

- 0.1 second is about the limit for having the user feel that the system is reacting instantaneously.
- 1 second is about the limit for the user's flow of thought to stay uninterrupted.
- 10 seconds is about the limit for keeping the user's attention focused.

About two decades later, Google published findings that half of all mobile browser users abandon a web page if it takes longer than three seconds to load. Forget three seconds, even a 100 ms increase in latency would result in a 1% decrease in sales for Amazon. That is a lot of lost revenue. Mitigating this by processing on the device instantaneously can make for rich and interactive UXs. Running deep learning models in real time, as is done with Snapchat Lenses, can increase engagement with users.

24/7 availability and reduced cloud costs

- Obviously, sending less data to the cloud equates to less computing costs for the developer, leading to monetary savings. This reduces scaling costs, as well, when an app gains traction and grows to a large user base. For the users, computation on the edge is helpful, too, because they don't need to worry about data plan costs. Additionally, processing locally means 24/7 availability, without the fear of losing connectivity.

Privacy

- For the user, local computation preserves privacy by not sharing the data externally, which could potentially be mined for user information. For the developer, this ensures less headache dealing with Personally Identifiable Information (PII). With the European Union's General Data Protection Regulation (GDPR) and other user data protection laws coming up around the world, this becomes even more important.

Hopefully, these arguments were convincing for why AI on mobile is important. For someone building any kind of serious application, the following are a few common questions to consider during the course of development:

- How do I convert my model to run on a smartphone?
- Will my model run on other platforms?
- How do I run my model fast?
- How do I minimize the size of the app?
- How do I ensure that my app doesn't drain the battery?
- How do I update my model without going through the roughly two-day app review process?
- How do I A/B test my models?
- Can I train a model on a device?
- How do I protect my intellectual property (i.e., model) from being stolen?

In the next three chapters, we look at how to run deep learning algorithms on

smartphones using different frameworks. In the process, we answer these questions as they come up.

In this chapter, we delve into the world of mobile AI for iOS devices. We first look at the general end-to-end software life cycle (shown in Figure 11-1) and see how the different parts fit in with one another. We explore the Core ML ecosystem, its history, and the features offered. Next, we deploy a real-time object classification app on an iOS device, and learn performance optimizations and benchmarks. And finally, we analyze some real-world apps built on Core ML.

Time to look at the big picture.

smartphones using different frameworks. In the process, we answer these questions as they come up.

In this chapter, we delve into the world of mobile AI for iOS devices. We first look at the general end-to-end software life cycle (shown in Figure 11-1) and see how the different parts fit in with one another. We explore the Core ML ecosystem, its history, and the features offered. Next, we deploy a real-time object classification app on an iOS device, and learn performance optimizations and benchmarks. And finally, we analyze some real-world apps built on Core ML.

Time to look at the big picture.

The Development Life Cycle for Artificial Intelligence on Mobile

Figure 11-1 depicts the typical life cycle for AI on mobile devices.

Figure 11-1. The mobile AI development life cycle

Let's look at the phases in Figure 11-1 a bit more closely:

1. *Collect data*: The data we collect should reflect the context in which the app would be used. Pictures taken by real users using smartphone cameras tend to be better training examples than pictures taken by professional photographers. We might not have this data on day one, but we can progressively collect more and more data as usage grows. A good starting point in many cases is to download images from search engines.

2. *Label data*: We need associated labels for the data samples that we want our model to predict. High-quality (i.e., correct) labels are vital to a good model.
3. *Train model*: We build the highest-accuracy neural network possible with the data and associated labels we have so far.
4. *Convert model*: Export the model from the training framework into the mobile-compatible framework.
5. *Optimize performance*: Due to the resource-constrained nature of a mobile device, it is crucial to make the model efficient for memory, energy, and processor usage.
6. *Deploy*: Add the model to the app and ship it to users.
7. *Monitor*: Track the app usage in the real world to find opportunities to improve further. Additionally, gather samples of real-word data from consenting users to feed to this life cycle, and then back to Step 1.

In the first portion of the book, we primarily explored phases 1, 2, and 3, and additionally general performance improvements. In this chapter, we focus on phases 4, 5, and 6. And in the next few chapters, we explore all of these phases in the context of mobile development.

TIP

Before getting the app in the hands of real users (who might hate it if the app doesn't perform as well as they expected), it's common practice to gather feedback through a process called dogfooding. As the saying goes: Eat Your Own Dog Food. The process involves having an inner circle of loyal users who get to test early releases, thereby identifying bugs before they go out to the general public. For AI development, this inner circle might also contribute data and assess the success of the AI models in the real world. And as the models improve, they can be deployed to a gradually increasing number of test users before finally deploying to the public.

Let's jump right in!

r

A Brief History of Core ML

Core ML provides one of the simplest ways to run a deep neural network for inference on an Apple device such as the iPhone and iPad as well as MacBook, Apple TV, and Apple Watch. In addition to being easy to use, it is also optimized for the underlying hardware architecture. Alternative frameworks have become a lot better in the past few years, but it's difficult to beat the simplicity and performance offered by Core ML.

Traditionally, to quickly run a CNN on an Apple device, developers *needed* to write on Metal, a library that was offered to game developers to better utilize the GPU. Unfortunately, developing on Metal was akin to writing in assembly language or CUDA code for an NVIDIA GPU. It was tedious, error prone, and difficult to debug. Few developers dared to tread that path. DeepLearningKit (December 2015) by Amund Tveit was one effort to build an abstraction over Metal for deploying CNNs.

At the Apple Worldwide Developers Conference (WWDC) 2016, the company announced Metal Performance Shaders (MPS), a framework built on top of Metal, as a high-performance library for optimizing graphics and certain compute operations. It abstracted away a lot of low-level details, giving basic building blocks such as Convolution, Pooling, and ReLU. It allowed developers to write deep neural networks by combining these operations in code. For anyone who comes from the Keras world, this is a familiar and not-so-daunting task. Unfortunately, there is a lot of bookkeeping involved when writing MPS code, because you need to manually keep track of input and output dimensions a each step of the process. As an example, the code sample released by Apple for running the InceptionV3 model for recognizing 1,000 object categories was well over 2,000 lines long, with most of them defining the network. Now imagine changing the model slightly during training, and then having to dig through all 2,000 lines of code to reflect the same update in iOS code. Forge (April 2017), a library by Matthijs Hollemans, was an effort to simplify development over MPS by reducing the boilerplate code necessary to get a model running.

All of these hardships went away when Apple announced Core ML at WWDC 2017. This included an inference engine on iOS and an open source Python

package called Core ML Tools to serialize CNN models from other frameworks like Keras and Caffe. The general workflow for building an app was: train a model in other packages, convert it to a *.mlmodel* file, and deploy it in an iOS app running on the Core ML platform.

Core ML supports importing a broad range of machine learning models built with first- and third-party frameworks and file formats. Figure 11-2 showcases a few of them (clockwise, starting in the upper left) such as TensorFlow, Keras, ONNX, scikit-learn, Caffe2, Apple's Create ML, LIBSVM, and TuriCreate (also from Apple). ONNX itself supports a large variety of frameworks, including PyTorch (Facebook), MXNet (Amazon), Cognitive Toolkit (Microsoft), PaddlePaddle (Baidu), and more, thereby ensuring compatibility with any major framework under the sky.

Figure 11-2. Frameworks compatible with Core ML for model interchange as of 2019

package called Core ML Tools to serialize CNN models from other frameworks like Keras and Caffe. The general workflow for building an app was: train a model in other packages, convert it to a *.mlmodel* file, and deploy it in an iOS app running on the Core ML platform.

Core ML supports importing a broad range of machine learning models built with first- and third-party frameworks and file formats. Figure 11-2 showcases a few of them (clockwise, starting in the upper left) such as TensorFlow, Keras, ONNX, scikit-learn, Caffe2, Apple's Create ML, LIBSVM, and TuriCreate (also from Apple). ONNX itself supports a large variety of frameworks, including PyTorch (Facebook), MXNet (Amazon), Cognitive Toolkit (Microsoft), PaddlePaddle (Baidu), and more, thereby ensuring compatibility with any major framework under the sky.

Figure 11-2. Frameworks compatible with Core ML for model interchange as of 2019

Alternatives to Core ML

Depending on the platform, there are a handful of options to achieve real-time predictions. These include general-purpose inference frameworks such as Core ML (from Apple), TensorFlow Lite (from Google), ML Kit (also from Google), and Fritz, as well as chip-specific accelerator frameworks including Snapdragon Neural Processing Engine (from Qualcomm) and Huawei AI Mobile Computing Platform (for Huawei's Neural Processing Unit). Table 11-1 presents a high-leve comparison of all these frameworks.

Table 11-1. A comparison of mobile device AI frameworks

Framework	Available for iOS	Available for Android	Dynamic updates	A/B testing	On-device training	Model encryption
Core ML	✓	—	✓	—	✓	—
TensorFlow Lite	✓	✓	—	—	Releases late 2019	—
ML Kit	✓	✓	✓	✓	—	—
Fritz	✓	✓	✓	✓	—	✓

Alternatives to Core ML

Depending on the platform, there are a handful of options to achieve real-time predictions. These include general-purpose inference frameworks such as Core ML (from Apple), TensorFlow Lite (from Google), ML Kit (also from Google), and Fritz, as well as chip-specific accelerator frameworks including Snapdragon Neural Processing Engine (from Qualcomm) and Huawei AI Mobile Computing Platform (for Huawei's Neural Processing Unit). Table 11-1 presents a high-level comparison of all these frameworks.

Table 11-1. A comparison of mobile device AI frameworks

Framework	Available for iOS	Available for Android	Dynamic updates	A/B testing	On-device training	Model encryption
Core ML	✓	—	✓	—	✓	—
TensorFlow Lite	✓	✓	—	—	Releases late 2019	—
ML Kit	✓	✓	✓	✓	—	—
Fritz	✓	✓	✓	✓	—	✓

TensorFlow Lite

In November 2017, Google announced an on-device inference engine called TensorFlow Lite with the intention of extending the TensorFlow ecosystem beyond just servers and PCs. Prior to this, the options within the TensorFlow ecosystem were porting the entire TensorFlow library, itself ported to iOS (which was heavy and slow), and later on, its slightly stripped-down version called TensorFlow Mobile (which was still pretty bulky).

TensorFlow Lite was rebuilt from the ground up targeting mobile and edge devices, optimizing for speed, model and interpreter size, and power consumption. It added support for GPU backend delegates, meaning that as long as there was GPU support implemented for a hardware platform, TensorFlow Lite could take advantage of the power of the GPU. On iOS, the GPU delegate uses Metal for acceleration. We discuss TensorFlow Lite at greater length in Chapter 13.

TensorFlow Lite

In November 2017, Google announced an on-device inference engine called TensorFlow Lite with the intention of extending the TensorFlow ecosystem beyond just servers and PCs. Prior to this, the options within the TensorFlow ecosystem were porting the entire TensorFlow library, itself ported to iOS (which was heavy and slow), and later on, its slightly stripped-down version called TensorFlow Mobile (which was still pretty bulky).

TensorFlow Lite was rebuilt from the ground up targeting mobile and edge devices, optimizing for speed, model and interpreter size, and power consumption. It added support for GPU backend delegates, meaning that as long as there was GPU support implemented for a hardware platform, TensorFlow Lite could take advantage of the power of the GPU. On iOS, the GPU delegate uses Metal for acceleration. We discuss TensorFlow Lite at greater length in Chapter 13.

ML Kit

ML Kit is a high-level library from Google that provides many computer vision, NLP, and AI functionalities out of the box, including the ability to run TensorFlow Lite models. Some of the features include face detection, barcode scanning, smart reply, on-device translation, and language identification. However, the main selling point of ML Kit is its integration with Google Firebase. Features offered by Firebase include dynamic model updates, A/B testing, and remote configuration-driven dynamic model selection (fancy words for choosing which model to use based on the customer). We explore ML Kit in greater detail in Chapter 13.

ML Kit

ML Kit is a high-level library from Google that provides many computer vision, NLP, and AI functionalities out of the box, including the ability to run TensorFlow Lite models. Some of the features include face detection, barcode scanning, smart reply, on-device translation, and language identification. However, the main selling point of ML Kit is its integration with Google Firebase. Features offered by Firebase include dynamic model updates, A/B testing, and remote configuration-driven dynamic model selection (fancy words for choosing which model to use based on the customer). We explore ML Kit in greater detail in Chapter 13.

Fritz

Fritz is a startup founded with the goal of making the end-to-end process of mobile inference easier. It bridges the gap between machine learning practitioners and mobile engineers by providing easy-to-use command-line tools On one side, it integrates training in Keras directly into the deployment pipeline, so a machine learning engineer could add a single line of Keras callback to deploy the model to users immediately after it finishes training. On the other side, a mobile engineer can benchmark the model without even needing to deploy to a physical device, simulating a model's performance virtually, assess a Keras model's compatibility with Core ML, and get analytics for each model. One unique selling point of Fritz is the model protection feature that prevents a model from deep inspection through obfuscation in the event that a phone is jailbroken.

Fritz

Fritz is a startup founded with the goal of making the end-to-end process of mobile inference easier. It bridges the gap between machine learning practitioners and mobile engineers by providing easy-to-use command-line tools. On one side, it integrates training in Keras directly into the deployment pipeline, so a machine learning engineer could add a single line of Keras callback to deploy the model to users immediately after it finishes training. On the other side, a mobile engineer can benchmark the model without even needing to deploy to a physical device, simulating a model's performance virtually, assess a Keras model's compatibility with Core ML, and get analytics for each model. One unique selling point of Fritz is the model protection feature that prevents a model from deep inspection through obfuscation in the event that a phone is jailbroken.

Apple's Machine Learning Architecture

To gain a better understanding of the Core ML ecosystem, it's useful to see the high-level picture of all the different APIs that Apple offers as well as how they fit with one another. Figure 11-3 gives us a look at the different components that make up Apple's machine learning architecture.

Figure 11-3. Different levels of APIs provided by Apple for app developers

Apple's Machine Learning Architecture

To gain a better understanding of the Core ML ecosystem, it's useful to see the high-level picture of all the different APIs that Apple offers as well as how they fit with one another. Figure 11-3 gives us a look at the different components that make up Apple's machine learning architecture.

Figure 11-3. Different levels of APIs provided by Apple for app developers

Domain-Based Frameworks

To simplify common tasks in machine learning without requiring domain expertise, Apple provides many APIs out of the box, in domains including Vision, Natural Language, Speech, and Sound Analysis. Table 11-2 gives a detailed outline of the functionalities available on Apple operating systems.

Table 11-2. Out-of-the-box machine learning functionality in Apple operating systems

Vision	Natural language	Other
• Facial landmark detection • Image similarity • Saliency detection • Optical character recognition • Rectangle detection • Face detection • Object classification • Barcode detection • Horizon detection • Human and animal detection • Object tracking (for video)	• Tokenization • Language identification • Parts of speech identification • Text embedding	• Speech recognition (on-device and on-cloud) • Sound classification

Domain-Based Frameworks

To simplify common tasks in machine learning without requiring domain expertise, Apple provides many APIs out of the box, in domains including Vision, Natural Language, Speech, and Sound Analysis. Table 11-2 gives a detailed outline of the functionalities available on Apple operating systems.

Table 11-2. Out-of-the-box machine learning functionality in Apple operating systems

Vision	Natural language	Other
• Facial landmark detection • Image similarity • Saliency detection • Optical character recognition • Rectangle detection • Face detection • Object classification • Barcode detection • Horizon detection • Human and animal detection • Object tracking (for video)	• Tokenization • Language identification • Parts of speech identification • Text embedding	• Speech recognition (on-device and on-cloud) • Sound classification

ML Framework

Core ML gives the ability to run inference on deep learning and machine learning models.

ML Framework

Core ML gives the ability to run inference on deep learning and machine learning models.

ML Performance Primitives

The following are some of the machine learning primitives in the Apple stack:

MPS

Provides low-level and high-performance primitives that utilize the GPU to aid running most CNN-based networks fast. And if Core ML does not support a model, MPS provides all of the building blocks to enable us to build them. Additionally, we might consider using MPS to hand roll a model for performance reasons (like guaranteeing that the model runs on the GPU).

Accelerate and Basic Neural Network Subroutine

Accelerate is Apple's implementation of the Basic Linear Algebra Subprogram (BLAS) library. It provides functionality for high-performance large-scale mathematical computations and image calculations, like Basic Neural Network Subroutine (BNNS), which helps to implement and run neural networks.

Now that we have seen how Core ML and the domain-specific APIs fit into the overall architecture, let's see how little work is needed to run a machine learning model using Core ML and the Vision framework on an iOS app.

TIP

Apple provides several downloadable models (Figure 11-4) for various computer-vision tasks, from classification to detecting objects (with bounding boxes), segmenting (identifying the pixels), depth estimation, and more. You can find them at *https://developer.apple.com/machine-learning/models/*.

For classification, you can find many pretrained Core ML models on Apple's Machine Learning website, including MobileNet, SqueezeNet, ResNet-50, and VGG16.

Figure 11-4. Ready-to-use models from the Apple Machine Learning website

Figure 11-4. Ready-to-use models from the Apple Machine Learning website

Building a Real-Time Object Recognition App

Although we don't intend to teach iOS development, we want to demonstrate running an object recognition model that classifies among 1,000 ImageNet categories in real time on an iOS device.

We will examine the minimal amount of code necessary to get this app running. The general outline is as follows:

1. Drag and drop an *.mlmodel* file to the Xcode project.
2. Load the model into a Vision container (`VNCoreMLModel`).
3. Create a request (`VNCoreMLRequest`) based on that container and provide a function that would be called when the request is completed.
4. Create a request handler that can process the request based on an image provided.
5. Execute the request and print out the results.

Let's take a look at the code to do this:

```
import CoreML
import Vision

// load the model
let model = try? VNCoreMLModel(for: Resnet50().model)!
// create a request with a callback
let classificationRequest = VNCoreMLRequest(model: model) {
    (request, error) in
    // print the results once the request is complete
    if let observations = request.results as? [VNClassificationObservation] {
        let results = observations
                      .map{"\($0.identifier) - \($0.confidence)"}
                      .joined(separator: "\n")
        print(results)
    }
}
// create a request handler taking an image as an argument
let requestHandler = VNImageRequestHandler(cgImage: cgImage)
// execute the request
try? requestHandler.perform([classificationRequest])
```

In this code, `cgImage` can be an image from a variety of sources. It can be a photo from the photo library, or from the web.

We can also power real-time scenarios using the camera and pass the individual camera frames into this function. A regular iPhone camera can shoot up to 60 frames per second (FPS).

By default, Core ML crops the image along the longer edge. In other words, if a model requires square dimensions, Core ML will extract the biggest square in the center of the image. This can be a source of confusion for developers who find the top and bottom strips of an image are being ignored for making predictions. Depending on the scenario, we might want to use `.centerCrop`, `.scaleFit`, or the `.scaleFill` option, as illustrated in Figure 11-5, such as the following:

```
classificationRequest.imageCropAndScaleOption = .scaleFill
```

Figure 11-5. How different scaling options modify the input image to Core ML models

Now that we have gone through the meat of it, what could be more fun? How about actually running it on a phone! We've made a complete app available on the book's GitHub website (see *http://PracticalDeepLearning.ai*) at *code/chapter-11*. With an iPhone or an iPad, we can deploy this pretty quickly and experiment with it, even without knowledge of iOS development. Here are the steps (note: this requires a Mac):

1. Download Xcode from the Apple developer website or the Mac App Store.
2. Plug in an iOS device. The phone needs to remain unlocked during deployment.

3. Change the current working directory to `CameraApp`:

```
$ cd code/chapter-11/CameraApp
```

4. Download the Core ML models from the Apple website using the available Bash script:

```
$ ./download-coreml-models.sh
```

5. Open the Xcode project:

```
$ open CameraApp.xcodeproj
```

6. In the Project Hierarchy Navigator, in the upper-left corner, click the CameraApp project, as shown in Figure 11-6, to open the Project Information view.

Figure 11-6. Project information view within Xcode

7. Because Xcode reserves a unique bundle identifier, use a unique name to identify the project.

8. Log in to an Apple account to let Xcode sign the app and deploy it to the device. Select a team to do the signing, as shown in Figure 11-7.

Figure 11-7. Select a team and let Xcode automatically manage code signing

9. Click the "Build and Run" button (the right-facing triangle) to deploy the app on the device, as shown in Figure 11-8. This should typically take around 30 to 60 seconds.

Figure 11-8. Select the device and click the "Build and Run" button to deploy the app

10. The device will not run the app right away, because it is not trusted. Go to Settings > General > Profiles and Device Management and select the row with your information on it, and then tap "Trust {your_email_id}," as shown in Figure 11-9.

Figure 11-9. Profiles and Device Management screen

11. On the home screen, find CameraApp and run the app.

Figure 11-10. Screenshot of the app

The output shows predictions of “notebook, notebook computer” with 84% confidence and “laptop, laptop computer” with 11% confidence.

That was all fun and good. Let’s now get down to more serious business: converting models from different frameworks to Core ML models.

TIP

A great thing about Xcode is that the model input and output parameters are shown when we load the *.mlmodel* file into Xcode, as demonstrated in Figure 11-11. This is especially helpful when we have not trained the model ourselves and don't necessarily want to write code (like `model.summary()` in Keras) to explore the model architecture.

Figure 11-11. Xcode model inspector showing the inputs and outputs of the MobileNetV2 model

TIP

A great thing about Xcode is that the model input and output parameters are shown when we load the *.mlmodel* file into Xcode, as demonstrated in Figure 11-11. This is especially helpful when we have not trained the model ourselves and don't necessarily want to write code (like `model.summary()` in Keras) to explore the model architecture.

Figure 11-11. Xcode model inspector showing the inputs and outputs of the MobileNetV2 model

Conversion to Core ML

In the code sample we built, you saw the *Inceptionv3.mlmodel* file. Ever wonder how that file came to be? Inception was, after all, trained by Google using TensorFlow. That file was converted from a *.pb* file into a Core ML model. We might similarly have models that need to be converted from Keras, Caffe or any other framework to Core ML. The following are a few tools that enable model conversion to Core ML.

- Core ML Tools (Apple): from Keras (*.h5*), Caffe (*.caffemodel*), as well as machine learning libraries such as LIBSVM, scikit-learn, and XGBoost.
- tf-coreml (Google): from TensorFlow (*.pb*).
- onnx-coreml (ONNX): from ONNX (*.onnx*).

We take a look at the first two converters in detail.

Conversion to Core ML

In the code sample we built, you saw the *Inceptionv3.mlmodel* file. Ever wonder how that file came to be? Inception was, after all, trained by Google using TensorFlow. That file was converted from a *.pb* file into a Core ML model. We might similarly have models that need to be converted from Keras, Caffe or any other framework to Core ML. The following are a few tools that enable model conversion to Core ML.

- Core ML Tools (Apple): from Keras (*.h5*), Caffe (*.caffemodel*), as well as machine learning libraries such as LIBSVM, scikit-learn, and XGBoost.
- tf-coreml (Google): from TensorFlow (*.pb*).
- onnx-coreml (ONNX): from ONNX (*.onnx*).

We take a look at the first two converters in detail.

Conversion from Keras

Core ML Tools aids the translation of Keras, ONNX, and other model formats tc the Core ML format (*.mlmodel*). Install the `coremltools` framework using `pip`:

```
$ pip install --upgrade coremltools
```

Now, let's see how we can use an existing Keras model and convert it to Core ML. The conversion is just a one-line command and then we can save the converted model, as shown here:

```
from tensorflow.keras.applications.resnet50 import ResNet50
model = ResNet50()

import coremltools
coreml_model = coremltools.converters.keras.convert(model)
coreml_model.save("resnet50.mlmodel")
```

That's it! It can't get any simpler. We discuss how to convert models with unsupported layers (such as MobileNet) in Chapter 12.

Conversion from Keras

Core ML Tools aids the translation of Keras, ONNX, and other model formats to the Core ML format (*.mlmodel*). Install the `coremltools` framework using `pip`:

```
$ pip install --upgrade coremltools
```

Now, let's see how we can use an existing Keras model and convert it to Core ML. The conversion is just a one-line command and then we can save the converted model, as shown here:

```
from tensorflow.keras.applications.resnet50 import ResNet50
model = ResNet50()

import coremltools
coreml_model = coremltools.converters.keras.convert(model)
coreml_model.save("resnet50.mlmodel")
```

That's it! It can't get any simpler. We discuss how to convert models with unsupported layers (such as MobileNet) in Chapter 12.

Conversion from TensorFlow

Apple recommends using `tf-coreml` (from Google) for converting from TensorFlow-based models. In the following steps, we convert a pretrained TensorFlow model to Core ML. This process is a bit more involved than the single line of code we saw earlier.

First, we perform the installation of `tfcoreml` using `pip`:

```
$ pip install tfcoreml --user --upgrade
```

To do the conversion, we need to know the first and last layers of the model. We can ascertain this by inspecting the model architecture using a model visualization tool like Netron. After loading the MobileNet model (*.pb* file) in Netron, we can visualize the entire model in the graphical interface. Figure 11-12 shows a small portion of the MobileNet model; in particular, the output layer.

Figure 11-12. Output layer of MobileNet as seen in Netron

We simply need to plug that into the following Python code as an argument and run it:

```
import tfcoreml as tf_converter
tf_converter.convert(tf_model_path = "input_model.pb",
     mlmodel_path = "output_model.mlmodel",
     output_feature_names = ["MobilenetV1/Predictions/Reshape_1:0"])
```

As the script runs, we see each of the operations from each layer being converted to their corresponding Core ML equivalents. When finished, we should find the *.mlmodel* exported in the directory.

Our Core ML model is now ready. Drag it into Xcode to test it.

```
import tfcoreml as tf_converter
tf_converter.convert(tf_model_path = "input_model.pb",
     mlmodel_path = "output_model.mlmodel",
     output_feature_names = ["MobilenetV1/Predictions/Reshape_1:0"])
```

As the script runs, we see each of the operations from each layer being converted to their corresponding Core ML equivalents. When finished, we should find the *.mlmodel* exported in the directory.

Our Core ML model is now ready. Drag it into Xcode to test it.

Dynamic Model Deployment

As developers, we will likely want to keep improving our models over time. After all, we'd want our users to have access to the latest and greatest model as quickly as we can hand it to them. One way to handle that is to send an update to the App Store every time we want to deploy a new model. That approach does not work very well, because we'd need to wait about two days for Apple's approval each time, potentially causing significant delays.

An alternate recommended approach would be to have the app dynamically download the *.mlmodel* file and compile it within the user's device. There are several reasons why we might want to try this approach:

- We want to regularly update the model, independent of the cadence of our App Store releases.
- We want to keep the app download size small and have the user eventually download only the models relevant for their usage scenarios. This will reduce storage needs and bandwidth costs. Apple places a limit of 200 MB on App Store downloads via cellular networks, so it's vital to remain under that limit to not lose out on potential downloads.
- We want to A/B test different models on different sets of users to further improve the quality of the model.
- We want to use different models for different user segments, regions, and locales.

The process to achieve this is rather simple: host the *.mlmodel* on a server and design our app to dynamically download the file. After the model is on the user's device, we can run `MLModel.compileModel` on that file to generate a compiled version of the model:

```
let compiledModelUrl = try MLModel.compileModel(at: downloadedModelUrl)
let model = try MLModel(contentsOf: compiledModelUrl)
```

Keep in mind that `compiledModelUrl` is an address to a temporary location. If

you want to keep the model on the device longer beyond a session, you must move it to permanent storage.

> **NOTE**
>
> Although we can manually do model management directly with Core ML, it would still involve writing a lot of boilerplate code, on the backend and on the device. We would need to manually manage the versions of each model, the file storage, and configuration infrastructure, as well as any errors in the process. This is where ML Kit and Fritz really shine by providing these features out of the box. We discuss this in more detail in Chapter 13.

you want to keep the model on the device longer beyond a session, you must move it to permanent storage.

NOTE

Although we can manually do model management directly with Core ML, it would still involve writing a lot of boilerplate code, on the backend and on the device. We would need to manually manage the versions of each model, the file storage, and configuration infrastructure, as well as any errors in the process. This is where ML Kit and Fritz really shine by providing these features out of the box. We discuss this in more detail in Chapter 13.

On-Device Training

So far, we have looked at scenarios that would be served well by a “one size fits all” neural network. However, some use cases would simply not work without personalizing the models. An example would be an app that organizes a user’s photo gallery based on recognizing the faces of people in each picture. Given that an average user’s phone is mostly filled with pictures of friends and family, a generic model trained on Danny DeVito’s and Tom Hanks’s faces would not be very useful for users (unless, of course, they belong to the DeVito or Hanks families).

A real-life example would be on the iOS system keyboard, which learns a user’s language patterns over time and begins to give more and more relevant suggestions for that user. And this becomes even more evident when that person uses slang words, nicknames, domain-specific terms, and so on, which might not be part of the common language dictionary. Personalized suggestions based on such data would be useless for any other user.

In these cases, we want to collect the data for that user and train a personalized model for that individual only. One way to accomplish this would be to collect and send the data to the cloud, train a new model there, and send back the updated model to the user. This approach reeks of scalability, cost, and privacy issues.

Instead, Core ML provides functionality for on-device training, so that user’s data never needs to leave the device. A Core ML model is updatable when its `isUpdatable` property is set to `true`. Additionally, the specific set of layers that need to be retrained (usually toward the tail end of the network) also need to have the same property set to `true`. Additional training parameters such as learning rate and optimizers can also be set on the model.

Even though training can consume GPU and Neural Processor Unit (NPU; more on these in Chapter 13) cycles, the training can be scheduled in the background (using the `BackgroundTasks` framework), even when the device is idle and charging, typically at night. This will have the least impact on UX.

To do on-device training, we can call the `MLUpdateTask` function, along with the

new data with which we will be retraining the model. We would also need to pass in the path for the newly updated model in the function call. After training is complete, the model at that path is ready to go:

```
let modelUrl = bundle.url(forResource: "MyClassifier",
                          withExtension: "mlmodelc")!
let updatedModelUrl = bundle.url(forResource: "MyClassifierUpdated",
                                 withExtension: "mlmodelc")!

let task = try MLUpdateTask(
    forModelAt: modelUrl,
    trainingData: trainData,
    configuration: nil,
    completionHandler: { [weak self] (updateContext) in
          self.model = updateContext.model
          updateContext.model.write(to: updatedModelUrl)
    })

task.resume()
```

new data with which we will be retraining the model. We would also need to pass in the path for the newly updated model in the function call. After training is complete, the model at that path is ready to go:

```
let modelUrl = bundle.url(forResource: "MyClassifier",
                          withExtension: "mlmodelc")!
let updatedModelUrl = bundle.url(forResource: "MyClassifierUpdated",
                                 withExtension: "mlmodelc")!

let task = try MLUpdateTask(
    forModelAt: modelUrl,
    trainingData: trainData,
    configuration: nil,
    completionHandler: { [weak self] (updateContext) in
          self.model = updateContext.model
          updateContext.model.write(to: updatedModelUrl)
    })

task.resume()
```

Federated Learning

On-device training is great—except for one downside: the generic global model does not get a chance to improve. Wouldn't it be great for a developer to somehow use that data being generated on each user's device to improve the global model without needing to transfer their data from their device? This is where *federated learning* comes in.

Federated learning is a collaborative distributed training process. It essentially takes on-device training one step further by sending the incremental updates (on the personalized models from users' devices) to the cloud and aggregating many of these over the entire user base, subsequently enriching the global model for everyone. Keep in mind that no user data is being transmitted here and it is not possible to reverse engineer the user data from the aggregated feature set. With this approach, we are able to respect our users' privacy while also benefiting everyone through collective participation.

On-device training is a vital stepping stone for federated learning. Even though we are not there yet, this is the direction in which the industry is moving. We can expect to see more support for federated learning over time.

TensorFlow Federated is one such implementation of federated learning, powering training for Google's GBoard, a keyboard app. Training on user devices occurs in the background when they are being charged.

Federated Learning

On-device training is great—except for one downside: the generic global model does not get a chance to improve. Wouldn't it be great for a developer to somehow use that data being generated on each user's device to improve the global model without needing to transfer their data from their device? This is where *federated learning* comes in.

Federated learning is a collaborative distributed training process. It essentially takes on-device training one step further by sending the incremental updates (on the personalized models from users' devices) to the cloud and aggregating many of these over the entire user base, subsequently enriching the global model for everyone. Keep in mind that no user data is being transmitted here and it is not possible to reverse engineer the user data from the aggregated feature set. With this approach, we are able to respect our users' privacy while also benefiting everyone through collective participation.

On-device training is a vital stepping stone for federated learning. Even though we are not there yet, this is the direction in which the industry is moving. We can expect to see more support for federated learning over time.

TensorFlow Federated is one such implementation of federated learning, powering training for Google's GBoard, a keyboard app. Training on user devices occurs in the background when they are being charged.

Performance Analysis

Writing prototypes is one thing. Making production-ready apps is another ball game entirely. There are several factors that influence the user’s experience, and it is important to understand the trade-offs. Some of these include the supported device models, minimum operating system (OS) version, processing frame rate, and choice of the deep learning model. In this section, we explore the impact tha some of these factors can have on the quality and performance of the product.

Performance Analysis

Writing prototypes is one thing. Making production-ready apps is another ball game entirely. There are several factors that influence the user's experience, and it is important to understand the trade-offs. Some of these include the supported device models, minimum operating system (OS) version, processing frame rate, and choice of the deep learning model. In this section, we explore the impact that some of these factors can have on the quality and performance of the product.

Benchmarking Models on iPhones

As is the case with benchmarking, it is best to perform experiments on publicly available models that we can download easily and get our hands dirty with!

First, we ran our real-time object classification app on multiple iPhones produced from 2013 to 2018. Table 11-3 catalogs the results of those experiments.

Table 11-3. Benchmarking inference time on different models on different iPhone versions

Device model			iPhone 5s	iPhone 6	iPhone 6s	iPhone 7+	iPhone X	iPhone XS	iPhone 11 Pro
Year released			**2013**	**2014**	**2015**	**2016**	**2017**	**2018**	**2019**
RAM			**1 GB**	**1 GB**	**2 GB**	**2 GB**	**2 GB**	**4 GB**	**4GB**
Processor chip			**A7**	**A8**	**A9**	**A10**	**A11**	**A12**	**A13**
Model	**Accuracy (%)**	**Size (MB)**	**FPS**	**FPS**	**FPS**	**FPS**	**FPS**	**FPS**	**FPS**
VGG-16	71	553	0.1	0.2	4.2	5.5	6.9	27.8	34.5
InceptionV3	78	95	1.4	1.5	9	11.1	12.8	35.7	41.7
ResNet-50	75	103	1.9	1.7	11.6	13.5	14.1	38.5	50
MobileNet	71	17	7.8	9	19.6	28.6	28.6	55.6	71.4
SqueezeNet	57	5	13.3	12.4	29.4	33.3	34.5	66.7	76.9

The difference between inference times in 2014 and 2015 is particularly striking What happened in 2015? If you guessed GPUs, you'd be right. The iPhone 6S introduced a dedicated GPU for the first time, powering features such as "Hey Siri."

BENCHMARKING METHODOLOGY

To do reproducible benchmarking, we took the code of the example app from earlier in the chapter, and simply swapped the *.mlmodel* file within Xcode with other pretrained models available from Apple. For each kind of model, we ran the app for a fixed amount of time and collected

measurements on execution time from the debug console. After running it on different iPhones, we averaged the results from the 6th prediction to the 20th prediction for each run and tabulated those results.

On the other hand, let's look at the device market share among iPhones in the release month of the iPhone XS (Sept. 2018) in the United States, as shown in Table 11-4. Note that releases typically happen in September of each year.

Table 11-4. US device market share among iPhones as of September 2018 (the release month of the iPhone XS; data from Flurry Analytics adjusted to exclude iPhone XS, XS Plus, and XS Max)

Year released	iPhone model	Percentage
2017	8 Plus	10.8%
2017	X	10.3%
2017	8	8.1%
2016	7	15.6%
2016	7 Plus	12.9%
2016	SE	4.2%
2015	6S	12.5%
2015	6S Plus	6.1%
2014	6	10.7%
2014	6 Plus	3.3%
2013	5S	3.4%
2013	5C	0.8%
2012	5	0.8%
2011	4S	0.4%
2010	4	0.2%

Deriving the cumulative percentages from Table 11-4, we get Table 11-5. This table expresses our potential market share based on the oldest device we choose to support. For example, iPhones released after September 2016 (two years

before September 2018) have a total market share of 61.9%.

Table 11-5. Cumulative market share of iPhones, year-on-year, going back in time

Years under	Cumulative percentage
1	29.2%
2	61.9%
3	80.5%
4	94.5%
5	98.7%
6	99.5%

Combining the benchmark and market share, there are some design choices and optimizations available to us:

Using a faster model

On an iPhone 6, VGG-16 runs about 40 times slower than MobileNet. On an iPhone XS, it's still about two times slower. Just choosing a more efficient model can have a dramatic impact on performance, often without the need to compromise an equivalent amount of accuracy. It's worth noting that MobileNetV2 and EfficientNet offer an even better combination of speed and accuracy.

Determining the minimum FPS to support

A feature to apply filters on a real-time camera feed will need to be able to process a high number of the frames delivered by the camera every second to ensure a smooth experience. In contrast, an app that processes a single image due to a user-initiated action does not need to worry about performance nearly as much. A lot of applications are going to be somewhere in between. It's important to determine the minimum necessary FPS and perform benchmarking similar to Table 11-5 to arrive at the best model(s) for the scenario across iPhone generations.

Batch processing

GPUs are really good at parallel processing. So, it comes as no surprise that

processing a batch of data is more efficient than performing inferences on each item individually. Core ML takes advantage of this fact by exposing batch processing APIs. Some user experiences, particularly those that are asynchronous and/or memory intensive, can vastly benefit from these batching APIs. For example, any bulk processing of photos in the photo gallery. Instead of processing one image at a time, we can send a list of images together to the batch API, so that Core ML can optimize the performance on the GPU.

Dynamic model selection

Depending on the use case, we might not care nearly as much about high levels of accuracy as we do about a smooth experience (at some minimum FPS threshold). In these scenarios, we might choose a lighter, less-accurate model on a slower, older device, and a larger, more-accurate model on faster, modern devices. This can be used in conjunction with model deployment via the cloud so that we don't needlessly increase the app size.

Using Sherlocking to our advantage

"Sherlocking" is a term used for when first-party vendors (Apple in particular) make third-party software obsolete by baking in a feature themselves. A good example of that is when Apple released the Flashlight feature within the iPhone rendering all third-party flashlight apps (whether paid or free) obsolete. For illustration here, take the example of a hypothetical app that released a face tracking feature in 2017. A year later, Apple added a more accurate face tracking API within Core ML for iOS 12. Because the neural networks for the face tracking come baked into the OS, the app could update its code to use the built-in API. However, because the API would still not be available for any users in iOS 11, the app could use a hybrid approach in which it would remain backward compatible by using the old code path for iOS 11. The app developer could then stop bundling the neural network into the app, thereby reducing its size, potentially by several megabytes, and dynamically download the older model for users of iOS 11.

Graceful degradation

Sometimes an older phone just cannot handle newer deep learning models due to performance needs. That's a perfectly reasonable situation to be in, particularly if the feature is cutting edge. In those scenarios, it's acceptable to

use one of two approaches. The first is to offload the computation to the cloud. This obviously comes at the cost of interactivity and privacy. The other alternative is to disable the feature for those users and display a message to them stating why it is not available.

use one of two approaches. The first is to offload the computation to the cloud. This obviously comes at the cost of interactivity and privacy. The other alternative is to disable the feature for those users and display a message to them stating why it is not available.

Measuring Energy Impact

In the previous chapters, we focused on hosting classifiers on servers. Although there are several factors that affect design decisions, energy consumption is ofte not among them. However, on the client side, battery power tends to be limited, and minimizing its consumption becomes a priority. How users perceive a product depends a lot on how much energy it consumes. Remember the good ol days when GPS was a battery hog. Many apps that required location access used to get tons of one-star ratings for eating up too much battery power. And we definitely don't want that.

> **NOTE**
>
> People are pretty generous with bad reviews when they realize an app is eating their battery faster than anticipated, as the reviews shown in Figure 11-13 demonstrate.

Figure 11-13. App Store reviews for YouTube and Snapchat that complain about heavy battery consumptio

Here we utilize the Energy Impact tab (Figure 11-14) in Xcode's Debug Navigator and generate Figure 11-15.

Figure 11-14. Xcode Debug Navigator tab

Figure 11-15. Xcode Energy Impact chart on an iPad Pro 2017 (note: this screenshot was taken at a different time than Figure 11-14, which is why the numbers are slightly different)

Figure 11-15 demonstrates that the energy impact of running the process on the iPad Pro 2017 is high. A big reason for this is that in our sample code, we were processing every single frame that we were receiving from the camera. This meant that each frame that the camera captured was being sent to the GPU for processing, resulting in higher energy consumption. In many real-world applications, it is not necessary to classify every single frame. Even processing every other frame can result in significant energy savings without generally affecting the UX. In the next section, we explore the relationship between the frame processing rate and the energy impact.

NOTE

The ratio of CPU to GPU usage is affected by a model's architecture, specifically the number of convolutional operations. Here we profile the average CPU and GPU utilization for different models (Figure 11-16). These numbers can be extracted from the Energy Impact chart in Xcode. Note that we'd ideally prefer models with more GPU utilization for performance efficiency.

Figure 11-16. Comparing CPU and GPU utilization for different models on iOS 11

Figure 11-16. Comparing CPU and GPU utilization for different models on iOS 11

Benchmarking Load

As you would expect, running CNN models in real time on each frame causes high GPU/CPU utilization, which in turn rapidly drains the battery. Users might also notice the phone heating up.

Instead of running an analysis on every frame, how about we skip a few frames? MobileNet takes 20 ms to analyze one frame on an iPad Pro 2017. So, it classifies about 50 FPS. Instead, if we run it at 1 FPS, the GPU utilization is reduced from 42% to a mere 7%—a reduction of more than 83%! For many applications, processing at 1 FPS might be sufficient while still keeping the device cool. For example, a security camera might perform reasonably well even with processing frames once every couple of seconds.

By varying the number of frames analyzed per second and measuring the percentage of GPU utilization, we observe a fascinating trend. It is evident from the graph in Figure 11-17 that the higher the FPS, the more the GPU is utilized. That has a significant impact on energy consumption. For apps that need to be run for longer periods of time, it might be beneficial to reduce the number of inferences per second.

Figure 11-17. Varying the FPS and analyzing the load on an iPad Pro 2017

The values shown in the graph were derived from the Core Animation Instrument of Xcode Instruments. Following is the process we used to generate our results:

1. From Xcode, click Product and then select Profile.
2. After the Instruments window appears, select the Core Animation Instrument, as shown in Figure 11-18.

Figure 11-18. The Instruments window in Xcode Instruments

3. Press the Record button to start running the app in the Profiling mode.
4. Wait a few seconds to begin collecting instrumentation data.
5. Measure the values in the GPU Hardware Utilization column (Figure 11-19).

Figure 11-19. An app being profiled live in the Core Animation instrument

So far in this chapter, we've explored techniques for making production-worthy apps using Core ML. In the next section, we discuss examples of such apps in the real world.

Figure 11-19. An app being profiled live in the Core Animation instrument

So far in this chapter, we've explored techniques for making production-worthy apps using Core ML. In the next section, we discuss examples of such apps in the real world.

Reducing App Size

App size can be crucially important to developers in certain markets. Apple does not allow apps larger than 200 MB to be downloaded via the cellular network. This becomes a crucial point of contention for apps that are used frequently on the go, such as ride-hailing apps like Uber and Lyft. Interesting tidbit here: Uber had to do some really stringent optimizations such as heavily reducing the number of Swift Optionals, Structs, and Protocols (which otherwise help with code maintainability) in order to bring the app binary size below the App Store limit (at the time, 150 MB). Without doing that, the company would have lost a lot of new users.

It's a given that any new AI functionality we introduce in the app will result in additional usage of storage. There are a few strategies that we can employ to reduce the impact of that.

NOTE

Segment, a San Francisco-based data analytics company, wanted to determine how much of an impact app size has on the number of installs. To experiment, the company purchased a mortgage calculator app that had a steady number of app downloads (approximately 50 per day). It took the app that was originally 3 MB in size and kept bloating it repeatedly with…well, images of Taylor Swift albums (hey, it was in the name of science and research!). Engineers observed significant drops in the number of daily installs as the company increased the app size. When the app crossed 100 MB (the maximum limit for cellular network downloads from the App Store at the time), the number of daily installs dropped a whopping 44%! Additionally, the app attracted several negative reviews with users expressing incredulity at the app size.

The moral of the story here is that the app size is far more crucial than we think, and we ought to be mindful of the amount of space our app consumes before releasing it out to the public.

Reducing App Size

App size can be crucially important to developers in certain markets. Apple does not allow apps larger than 200 MB to be downloaded via the cellular network. This becomes a crucial point of contention for apps that are used frequently on the go, such as ride-hailing apps like Uber and Lyft. Interesting tidbit here: Uber had to do some really stringent optimizations such as heavily reducing the number of Swift Optionals, Structs, and Protocols (which otherwise help with code maintainability) in order to bring the app binary size below the App Store limit (at the time, 150 MB). Without doing that, the company would have lost a lot of new users.

It's a given that any new AI functionality we introduce in the app will result in additional usage of storage. There are a few strategies that we can employ to reduce the impact of that.

NOTE

Segment, a San Francisco-based data analytics company, wanted to determine how much of an impact app size has on the number of installs. To experiment, the company purchased a mortgage calculator app that had a steady number of app downloads (approximately 50 per day). It took the app that was originally 3 MB in size and kept bloating it repeatedly with…well, images of Taylor Swift albums (hey, it was in the name of science and research!). Engineers observed significant drops in the number of daily installs as the company increased the app size. When the app crossed 100 MB (the maximum limit for cellular network downloads from the App Store at the time), the number of daily installs dropped a whopping 44%! Additionally, the app attracted several negative reviews with users expressing incredulity at the app size.

The moral of the story here is that the app size is far more crucial than we think, and we ought to be mindful of the amount of space our app consumes before releasing it out to the public.

Avoid Bundling the Model

If possible, we should avoid bundling the model with the App Store binary. The same amount of data needs to be downloaded anyway, so as long as the UX is not affected, we should delay model download to when the feature is actually going to be used. Additionally, we should prefer to download the model when on WiFi in order to preserve cellular bandwidth. Microsoft Translator and Google Translate implement a form of this where they do only cloud-based translations at the outset. Knowing that travelers use these apps a lot (where they might not have good internet access), they also offer an offline mode where the required language models are downloaded at the user’s prompt in the background.

Avoid Bundling the Model

If possible, we should avoid bundling the model with the App Store binary. The same amount of data needs to be downloaded anyway, so as long as the UX is not affected, we should delay model download to when the feature is actually going to be used. Additionally, we should prefer to download the model when on WiFi in order to preserve cellular bandwidth. Microsoft Translator and Google Translate implement a form of this where they do only cloud-based translations at the outset. Knowing that travelers use these apps a lot (where they might not have good internet access), they also offer an offline mode where the required language models are downloaded at the user’s prompt in the background.

Use Quantization

As we discussed in Chapter 6, quantization is a good strategy to reduce model size drastically while still preserving performance. Essentially, it reduces 32-bit floating-point weights down to 16-bit floating point, and 8-bit integers all the way down to 1-bit. We definitely won't recommend going below 8 bits, though, due to the resulting loss in accuracy. We can achieve quantization with Core ML Tools for Keras models in just a couple of lines:

```
import coremltools

model_spec = coremltools.utils.load_spec("MyModel.mlmodel")

# 16-bit conversion
model_fp16_spec =
coremltools.utils.convert_neural_network_spec_weights_to_fp16(model_spec)
coremltools.utils.save_spec(model_fp16_spec, "MyModel_FP16.mlmodel")

# 8-bit or lower quantization
num_bits = 8
model_quant_spec =
coremltools.models.neural_network.quantization_utils.quantize_weights(model_spec,
num_bits, "linear")
coremltools.utils.save_spec(model_quant_spec, "MyModel_Quant.mlmodel")
```

To illustrate the impact of quantization on a challenging dataset where small modifications could lead to drastic changes, we chose to build a classifier for Oxford's 102 Category Flower Dataset with Keras (roughly 14 MB in size), and quantized it to different bit representations while measuring its accuracy and decreasing its size. To measure the change in predictions, we compare the percent match between the full-precision model and the quantized model. We tested three quantization modes.

- Simple `linear` quantization, which we described in Chapter 6. The intervals are distributed equally in this strategy.
- Linear quantization using a lookup table, or `linear_lut`. In this technique, the intervals are distributed unequally, with denser areas getting smaller anc more intervals, whereas sparser areas would have fewer and larger intervals. Because these intervals are unequal, they need to be stored in the

lookup table, rather than being directly computed with simple arithmetic.

- Lookup table generated by *k*-means, or `kmeans_lut`, which is often used in nearest neighbor classifiers.

Table 11-6 shows our observations.

Table 11-6. Quantization results for different target bit sizes and different quantization modes

		Percent match with 32-bit results		
Quantized to	**Percent size reduction (approx.)**	**`linear`**	**`linear_lut`**	**`kmeans_lut`**
16-bit	50%	100%	100%	100%
8-bit	75%	88.37%	80.62%	98.45%
4-bit	88%	0%	0%	81.4%
2-bit	94%	0%	0%	10.08%
1-bit	97%	0%	0%	7.75%

This gives us a few insights.

- Lowering representation to 16-bits has no effect at all on accuracy. We can essentially halve the model size with no perceivable difference in accuracy.
- *k*-means when used for building the lookup table outperforms simple linear division methods. Even at 4-bit, only 20% accuracy is lost, which is remarkable.
- Going to 8-bit quantized models gives 4 times smaller modes with little loss in accuracy (especially for `kmeans_lut` mode).
- Quantizing to lower than 8-bits results in a drastic drop in accuracy, particularly with the linear modes.

lookup table, rather than being directly computed with simple arithmetic.

- Lookup table generated by *k*-means, or `kmeans_lut`, which is often used in nearest neighbor classifiers.

Table 11-6 shows our observations.

Table 11-6. Quantization results for different target bit sizes and different quantization modes

		Percent match with 32-bit results		
Quantized to	**Percent size reduction (approx.)**	**`linear`**	**`linear_lut`**	**`kmeans_lut`**
16-bit	50%	100%	100%	100%
8-bit	75%	88.37%	80.62%	98.45%
4-bit	88%	0%	0%	81.4%
2-bit	94%	0%	0%	10.08%
1-bit	97%	0%	0%	7.75%

This gives us a few insights.

- Lowering representation to 16-bits has no effect at all on accuracy. We can essentially halve the model size with no perceivable difference in accuracy.
- *k*-means when used for building the lookup table outperforms simple linear division methods. Even at 4-bit, only 20% accuracy is lost, which is remarkable.
- Going to 8-bit quantized models gives 4 times smaller modes with little loss in accuracy (especially for `kmeans_lut` mode).
- Quantizing to lower than 8-bits results in a drastic drop in accuracy, particularly with the linear modes.

Use Create ML

Create ML is a tool from Apple to train models by simply dragging and dropping data into a GUI application on the Mac. It provides several templates including object classification/detection, sound classification, and text classification, among others, putting training AI in the hands of novices without requiring any domain expertise. It uses transfer learning to tune only the handful of layers that will be necessary for our task. Because of this, the training process can be completed in just a few minutes. The OS ships with the bulk of the layers (that are usable across several tasks), whereas the task-specific layers can be shipped as part of the app. The resulting exported model ends up being really small (as little as 17 KB, as we will see shortly). We get hands-on with Create ML in the next chapter.

Use Create ML

Create ML is a tool from Apple to train models by simply dragging and dropping data into a GUI application on the Mac. It provides several templates including object classification/detection, sound classification, and text classification, among others, putting training AI in the hands of novices without requiring any domain expertise. It uses transfer learning to tune only the handful of layers that will be necessary for our task. Because of this, the training process can be completed in just a few minutes. The OS ships with the bulk of the layers (that are usable across several tasks), whereas the task-specific layers can be shipped as part of the app. The resulting exported model ends up being really small (as little as 17 KB, as we will see shortly). We get hands-on with Create ML in the next chapter.

Case Studies

Let’s take a look at some real-world examples that use Core ML for mobile inferences.

Case Studies

Let’s take a look at some real-world examples that use Core ML for mobile inferences.

Magic Sudoku

Soon after the launch of ARKit on iOS in 2017, the Magic Sudoku app from Hatchlings, a game and mobile app startup, came out as one of the break-out hits. Simply point the phone at a Sudoku puzzle, and the app will show a solved Sudoku right on the piece of paper. As we might have guessed at this point, the system is using Core ML to run a CNN-based digit recognizer. The system goes through the following steps:

1. Use ARKit to get camera frames.
2. Use iOS Vision Framework to find rectangles.
3. Determine whether it is a Sudoku grid.
4. Extract 81 squares from the Sudoku image.
5. Recognize digits in each square using Core ML.
6. Use a Sudoku solver to finish the empty squares.
7. Project the finished Sudoku on the surface of original paper using ARKit, as shown in Figure 11-20.

Figure 11-20. Step-by-step solution to solving ARKit (image source)

The Hatchlings team first started with using the MNIST digit recognition model, which mostly consists of handwritten digits. But this was not robust for printed fonts. The team then photographed thousands of pages of Sudoku books, extracting the squares using its pipeline until it had a large number of images of

individual digits in a variety of fonts. To label this data, the team requested its fans classify each item as 0 through 9 and empty classes. Within 24 hours, the team got 600,000 digits scanned. The next step was to train a custom CNN, which needed to work fast, because the system needed to classify 81 square images. Deploying this model using Core ML, the app launched and become an overnight hit.

Running out with new users in the wild brought new cases not previously anticipated. Because most users do not have a Sudoku puzzle in front of them, they often search for Sudoku on computer screens, and this was difficult for the model to always recognize precisely. Additionally, due to a fixed focal length limitation of ARKit, the input image could be slightly blurry. To remedy this, the Hatchlings team collected additional examples of photos of computer screens with puzzles, blurred them slightly, and trained a new CNN with the additional data. With an App Store update, the whole experience became much more robust. In summary, while launching an app or a service, the app builder needs to constantly learn from new scenarios not previously anticipated.

individual digits in a variety of fonts. To label this data, the team requested its fans classify each item as 0 through 9 and empty classes. Within 24 hours, the team got 600,000 digits scanned. The next step was to train a custom CNN, which needed to work fast, because the system needed to classify 81 square images. Deploying this model using Core ML, the app launched and become an overnight hit.

Running out with new users in the wild brought new cases not previously anticipated. Because most users do not have a Sudoku puzzle in front of them, they often search for Sudoku on computer screens, and this was difficult for the model to always recognize precisely. Additionally, due to a fixed focal length limitation of ARKit, the input image could be slightly blurry. To remedy this, the Hatchlings team collected additional examples of photos of computer screens with puzzles, blurred them slightly, and trained a new CNN with the additional data. With an App Store update, the whole experience became much more robust. In summary, while launching an app or a service, the app builder needs to constantly learn from new scenarios not previously anticipated.

Seeing AI

Seeing AI is a talking camera app from Microsoft Research designed for the blind and low-vision community. It uses computer vision to describe people, text, handwriting, objects, currency, and more through spoken audio. Much of the image processing happens locally on the device, using Core ML. One of the core user needs is to recognize products—this can usually be determined by scanning close-ups of barcodes. But for a blind user, the location of a barcode is unknown, making most barcode apps difficult to use. To solve this, the team built a custom CNN trained with images containing barcodes at a variety of angles, sizes, lighting, and orientation. The user now attempts to rotate the objec in front of the iPhone, and when the CNN classifies the presence of a barcode (running in real time, frame by frame), it signals with an audible beep. The rate of beeping directly correlates with the area of the barcode visible to the camera. As the blind users begin to bring the barcode closer, it beeps faster. When it is close enough for the barcode reading library to be able to clearly see the barcode, the app decodes the Universal Product Code and speaks the product name. In the past, blind users typically had to purchase and carry around bulky laser barcode scanners, which usually cost more than $1,300. In fact, a charity raised millions to donate these hardware barcode readers to those who needed them. Now, deep learning can solve this problem for free. This is a good example of mixing computer vision and UX to solve a real-world problem.

Seeing AI

Seeing AI is a talking camera app from Microsoft Research designed for the blind and low-vision community. It uses computer vision to describe people, text, handwriting, objects, currency, and more through spoken audio. Much of the image processing happens locally on the device, using Core ML. One of the core user needs is to recognize products—this can usually be determined by scanning close-ups of barcodes. But for a blind user, the location of a barcode is unknown, making most barcode apps difficult to use. To solve this, the team built a custom CNN trained with images containing barcodes at a variety of angles, sizes, lighting, and orientation. The user now attempts to rotate the object in front of the iPhone, and when the CNN classifies the presence of a barcode (running in real time, frame by frame), it signals with an audible beep. The rate of beeping directly correlates with the area of the barcode visible to the camera. As the blind users begin to bring the barcode closer, it beeps faster. When it is close enough for the barcode reading library to be able to clearly see the barcode, the app decodes the Universal Product Code and speaks the product name. In the past, blind users typically had to purchase and carry around bulky laser barcode scanners, which usually cost more than $1,300. In fact, a charity raised millions to donate these hardware barcode readers to those who needed them. Now, deep learning can solve this problem for free. This is a good example of mixing computer vision and UX to solve a real-world problem.

HomeCourt

For anything in life, regular practice is a must if we want to get better at something, be it writing, playing a musical instrument, or cooking. However, the quality of practice is far more important than quantity. Using data to support our practice and monitor our progress does wonders to the rate at which we acquire a skill. This is exactly what NEX Team, a San Jose, California-based startup, set out to do for basketball practice with the help of its HomeCourt app. Running the app is easy: set it up on the ground or a tripod, point the camera at the basketball court, and hit record.

The app runs an object detector in real time on top of Core ML to track the ball, the people, and the basketball hoop. Among these people, the big question remains: who shot the ball? When the ball goes near the hoop, the app rewinds the video to identify the player who shot the ball. Then, it performs human pose estimation on the player to track the player's movements. If this wasn't impressive enough, it uses geometric transformations to convert this 3D scene of the court into a 2D map (as shown in the upper right of Figure 11-21) to track the locations from which the player shot the ball. The important thing to note here is that multiple models are being run simultaneously. Based on the tracked objects and the player's body position, joints, and so on, the app provides the player with stats and visualizations for the release height of each throw, release angle, release position, release time, player speed, and more. These stats are important because they have a high correlation with a successful attempt. Players are able to record an entire game as they play and then go home and analyze each shot so that they can determine their weak areas in order to improve for the next time.

In just under two years of existence, this tiny startup attracted hundreds of thousands of users as word of mouth spread, from amateurs to professionals. Even the National Basketball Association (NBA) partnered with NEX Team to help improve their players' performance using HomeCourt. All of this happened because they saw an unmet need and came up with a creative solution using deep learning.

Figure 11-21. The HomeCourt app tracking a player's shots live as they are playing

Figure 11-21. The HomeCourt app tracking a player's shots live as they are playing

InstaSaber + YoPuppet

Do you know the largest profit maker in the *Star Wars* franchise? It's not the movie ticket sales, and it's definitely not from selling DVDs. Here's a hint: it rhymes with merchandise. Lucas Film (and now Disney) made a lot of money selling R2D2 toys and Chewbacca costumes. Though the all-time favorite still remains the dear lightsaber. However, instead of looking cool like in the movies, the merchandise lightsabers are mostly made of plastic and frankly don't look as sci-fi. Plus, you swing it around a few times and there's a good chance you'll accidentally whack someone in the face.

Hart Woolery, the founder of 2020CV, decided to change that by bringing Hollywood-level VFX to our phone with the InstaSaber app. Simply roll up a sheet of paper, grip it, and point your phone's camera toward it and watch it transform into a glowing lightsaber, as illustrated in Figure 11-22. Wave your hand and not just watch it track it realistically in real time, but also hear the same sound effects as when Luke fought his father, (spoiler alert!) Darth Vader.

Taking the tracking magic to the next level, he built YoPuppet, which tracks the joints in hands in real time to build virtual puppets that mimic the motion of the hand. The suspension of disbelief works only if it is truly real time with no lag, is accurate, and realistic looking.

In addition to being realistic, these apps were a lot of fun to play with. No wonder InstaSaber became an insta-hit with millions of viral views online and in the news. The AI potential even got billionaire investor Mark Cuban to say, "you've got a deal" and invest in 2020CV.

Figure 11-22. Screenshots of InstaSaber and YoPuppet

FROM THE CREATOR'S DESK

By Hart Woolery, founder and CEO of 2020CV

Since starting 2020CV a few years ago, I've developed a number of apps that attempt to push the boundaries of these special effects. I will talk about two: InstaSaber and YoPuppet. InstaSaber turns a rolled-up sheet of paper into a virtual lightsaber, and YoPuppet transforms the user's hand into a virtual hand puppet. In both cases, I am using real-time camera frames as inputs to my models, and returning normalized x-y coordinates of feature locations inside those images. I got the idea for InstaSaber after attempting to track a handheld marker, because I was struggling at the time to track hands by themselves. The initial challenge—as is generally the case for most machine learning models—is gathering the data, labeling the data, and making sure you have enough data to create a generalized solution.

Just how many labeled images do you need for a machine learning model that can decode an object reliably with multiple variables like lighting, skin tone, and occlusion? I think the answer depends largely on the problem, but

at a minimum, it will generally be at least 10,000. Assuming it could take 30 seconds or more per image to label it correctly, that is a lot of time being invested. The simplest workaround is to augment your data (i.e., affine transforms, cropping, and color/lighting manipulation). While this turns a finite amount of data into a nearly infinite set of variants, your model will still be constrained by the quality of your initial set. While I manually labeled images for InstaSaber, I found it far more efficient to synthetically generate images of hands in various positions and environments for YoPuppet (Figure 11-23). Once you are able to train and iterate on your model, the next challenge is getting them to perform well within the constraints of a mobile device.

Figure 11-23. Real-time hand pose estimation within YoPuppet

It’s fairly easy to get a machine learning model running on a mobile device these days with the advent of such frameworks as Core ML. However, getting a model to run in real time and deliver a great user experience—without melting the user’s phone—is challenging. In the case of YoPuppet

and InstaSaber, I also had to find the initial location of the object, and track it in real time so I could crop a square out of the full camera image, scale it down, and pass it to my model. One of the tricks I use to maintain a smooth frame rate is buffering the images so that the CPU and GPU can respectively preprocess and run inference on the images in tandem.

Once your app is published and out in the wild, the real-world feedback will often make you realize there are a lot of issues with your model you may have never considered (the bias is real!). For example, one user said that painted fingernails seemed to be causing issues with hand detection. Fortunately, this kind of feedback only helps you further generalize your model and make it more robust to variations in your objects. To be certain, the process of delivering a great experience for all your users is a never-ending challenge, but the reward is that you will be a pioneer in an emerging area of technology.

and InstaSaber, I also had to find the initial location of the object, and track it in real time so I could crop a square out of the full camera image, scale it down, and pass it to my model. One of the tricks I use to maintain a smooth frame rate is buffering the images so that the CPU and GPU can respectively preprocess and run inference on the images in tandem.

Once your app is published and out in the wild, the real-world feedback will often make you realize there are a lot of issues with your model you may have never considered (the bias is real!). For example, one user said that painted fingernails seemed to be causing issues with hand detection. Fortunately, this kind of feedback only helps you further generalize your model and make it more robust to variations in your objects. To be certain, the process of delivering a great experience for all your users is a never-ending challenge, but the reward is that you will be a pioneer in an emerging area of technology.

Summary

In this chapter, we went on a whirlwind tour of the world of Core ML, which provides an inference engine for running machine learning and deep learning algorithms on an iOS device. Analyzing the minimal code needed for running a CNN, we built a real-time object recognition app that classifies among 1,000 ImageNet categories. Along the way, we discussed some useful facts about model conversion. Moving to the next level, we learned about practical techniques like dynamic model deployment and on-device training, while also benchmarking different deep learning models on various iOS devices, providing a deeper understanding of the battery and resource constraints. We also looked a how we can optimize the app size using model quantization. Finally, to take some inspiration from the industry, we explored real-life examples where Core ML is being used in production apps.

In the next chapter, we build an end-to-end application by training using Create ML (among other tools), deploying our custom trained classifier, and running it using Core ML.

Summary

In this chapter, we went on a whirlwind tour of the world of Core ML, which provides an inference engine for running machine learning and deep learning algorithms on an iOS device. Analyzing the minimal code needed for running a CNN, we built a real-time object recognition app that classifies among 1,000 ImageNet categories. Along the way, we discussed some useful facts about model conversion. Moving to the next level, we learned about practical techniques like dynamic model deployment and on-device training, while also benchmarking different deep learning models on various iOS devices, providing a deeper understanding of the battery and resource constraints. We also looked at how we can optimize the app size using model quantization. Finally, to take some inspiration from the industry, we explored real-life examples where Core ML is being used in production apps.

In the next chapter, we build an end-to-end application by training using Create ML (among other tools), deploying our custom trained classifier, and running it using Core ML.

Chapter 12. Not Hotdog on iOS with Core ML and Create ML

"I'm a rich," said Jian-Yang, a newly minted millionaire in an interview with Bloomberg (Figure 12-1). What did he do? He created the Not Hotdog app (Figure 12-2) and made the world "a better place."

Figure 12-1. Jian-Yang being interviewed by Bloomberg News after Periscope acquires his "Not Hotdog" technology (image source: From HBO's Silicon Valley)

To the few of us who may be confused (including a third of the authors of this book), we are making a reference to HBO's *Silicon Valley*, a show in which one of the characters is tasked with making SeeFood—the "Shazam for food." It was meant to classify pictures of food and give recipes and nutritional information. Hilariously, the app ends up being good only for recognizing hot dogs. Anything else would be classified as "Not Hotdog."

There are a few reasons we chose to reference this fictitious app. It's very much a part of popular culture and something many people can easily relate to. It's an exemplar: easy enough to build, yet powerful enough to see the magic of deep

learning in a real-world application. It is also very trivially generalizable to recognize more than one class of items.

Figure 12-2. The Not Hotdog app in action (image source: Apple App Store listing for the Not Hotdog app)

In this chapter, we work through a few different approaches to building a Not Hotdog clone. The general outline of the end-to-end process is as follows:

1. Collect relevant data.
2. Train the model.
3. Convert to Core ML.
4. Build the iOS app.

Table 12-1 presents the different options available for steps 1 through 3. Further

along in the chapter, we do a deep dive into each of them.

Table 12-1. Various approaches to getting a model ready for mobile deployment, right from scratch

Data collection	Training mechanism	Model conversion
• Find or collect a dataset • Fatkun Chrome browser extension • Web scraper using Bing Image Search API	• Web-based GUI: CustomVision.ai, IBM Watson, Clarifai, Google AutoML • Create ML • Fine-tune using any framework of choice like Keras	• Create ML, CustomVision.ai, and other GUI tools generate *.mlmodel*. • For Keras, use Core ML Tools. • For TensorFlow trained models, use `tf-coreml`.

Let's dive right in!

)

along in the chapter, we do a deep dive into each of them.

Table 12-1. Various approaches to getting a model ready for mobile deployment, right from scratch

Data collection	Training mechanism	Model conversion
• Find or collect a dataset • Fatkun Chrome browser extension • Web scraper using Bing Image Search API	• Web-based GUI: CustomVision.ai, IBM Watson, Clarifai, Google AutoML • Create ML • Fine-tune using any framework of choice like Keras	• Create ML, CustomVision.ai, and other GUI tools generate *.mlmodel*. • For Keras, use Core ML Tools. • For TensorFlow trained models, use `tf-coreml`.

Let’s dive right in!

Collecting Data

To begin solving any computer-vision task using deep learning, we first need to have a dataset of images to train on. In this section, we use three different approaches to collecting the images of the relevant categories in increasing order of time required, from minutes to days.

Collecting Data

To begin solving any computer-vision task using deep learning, we first need to have a dataset of images to train on. In this section, we use three different approaches to collecting the images of the relevant categories in increasing order of time required, from minutes to days.

Approach 1: Find or Collect a Dataset

The fastest way to get our problem solved is to have an existing dataset in hand. There are tons of publicly available datasets for which a category or a subcategory might be relevant to our task. For example, Food-101 (*https://www.vision.ee.ethz.ch/datasets_extra/food-101/*) from ETH Zurich contains a class of hot dogs. Alternatively, ImageNet contains 1,257 images of hot dogs. We can use a random sample of images from the remaining classes as "Not Hotdog."

To download images from a particular category, you can use the ImageNet-Utils tool:

1. Search for the relevant category on the ImageNet website; for example, "Hot dog."
2. Note the `wnid` (WordNet ID)in the URL: *http://image-net.org/synset?wnid=n07697537*.
3. Clone the ImageNet-Utils repository:

```
$ git clone --recursive
        https://github.com/tzutalin/ImageNet_Utils.git
```

4. Download the images for the particular category by specifying the `wnid`:

```
$ ./downloadutils.py --downloadImages --wnid n07697537
```

In case we can't find a dataset, we can also build our own dataset by taking pictures ourselves with a smartphone. It's essential that we take pictures representative of how our application would be used in the real world. Alternatively, crowdsourcing this problem, like asking friends, family, and coworkers, can generate a diverse dataset. Another approach used by large companies is to hire contractors who are tasked with collecting images. For example, Google Allo released a feature to convert selfies into stickers. To build it, they hired a team of artists to take an image and create the corresponding sticker so that they could train a model on it.

NOTE

Make sure to check the license under which the images in the dataset are released. It’s best to use images released under permissive licenses such as Creative Commons.

NOTE

Make sure to check the license under which the images in the dataset are released. It’s best to use images released under permissive licenses such as Creative Commons.

Approach 2: Fatkun Chrome Browser Extension

There are several browser extensions that allow us to batch download multiple images from a website. One such example is the Fatkun Batch Download Image, a browser extension available on the Chrome browser.

We can have the entire dataset ready in the following short and quick steps.

1. Add the extension to our browser.
2. Search for the keyword either on Google or Bing Image search.
3. Select the appropriate filter for image licenses in the search settings.
4. After the page reloads, scroll to the bottom of it a few times repeatedly to ensure more thumbnails are loaded on the page.
5. Open the extension and select “This Tab” option, as demonstrated in Figure 12-3.

,

Figure 12-3. Bing Search results for "hot dog"

6. Notice that all the thumbnails are selected by default. At the top of the screen, click the Toggle button to deselect all the thumbnails and select only the ones we need. We can set the minimum width and height to be 224 (most pretrained models take 224x224 as input size).

Figure 12-4. Selecting images through the Fatkun extension

7. In the upper-right corner, click Save Image to download all of the selected thumbnails to our computer.

NOTE

Note that the images shown in the screenshots are iconic images (i.e., the main object is in direct focus with a clean background). Chances are that using such images exclusively in our model will cause it to not generalize well to real-world images. For example, in images with a clean white background (like on an ecommerce website), the neural network might incorrectly learn that white background equals hot dog. Hence, while performing data collection, ensure that your training images are representative of the real world.

TIP

For the negative class of “Not Hotdog,” we want to collect random images that are abundantly available. Additionally, collect items that look similar to a hot dog but are not; for example, a submarine sandwich, bread, plate, hamburgers, and so on.

An absence of commonly co-occurring items with hot dogs like plates with food, tissues, ketchup bottles or packets, can mistakenly lead the model to think that those are the real hot dogs. So be sure to add these to the negative class.

When you install a browser extension like Fatkun, it will request permissions to read and modify data on all the websites we visit. It might be a good idea to disable the extension when you’re not using it for

downloading images.

downloading images.

Approach 3: Web Scraper Using Bing Image Search API

For building larger datasets, using Fatkun to collect images can be a tedious process. Additionally, images returned by the Fatkun browser extension are thumbnails and not the original size images. For large-scale image collections, we can use an API for searching images, like the Bing Image Search API, where we can establish certain constraints such as the keyword, image size, and license Google used to have the Image Search API, but it was discontinued in 2011.

Bing's Search API is an amalgamation of its AI-based image understanding and traditional information retrieval methods (i.e., using tags from fields like "alt-text," "metadata," and "caption"). Many times, we can end up with some numbe of irrelevant images because of misleading tags from these fields. As a result, we want to manually parse the collected images to make sure that they are actually relevant to our task.

When we have a very large image dataset, it can be a daunting task to have to go through it manually and filter out all of the poor training examples. It's easier to approach this in an iterative manner, slowly improving the quality of the training dataset with each iteration. Here are the high-level steps:

1. Create a subset of the training data by manually reviewing a small number of images. For example, if we have 50k images in our original dataset, we might want to manually select around 500 good training examples for the first iteration.
2. Train the model on those 500 images.
3. Test the model on the remaining images and get the confidence value for each image.
4. Among the images with the least confidence values (i.e., often mispredictions), review a subset (say, 500) and discard the irrelevant images. Add the remaining images from this subset to the training set.
5. Repeat steps 1 through 4 for a few iterations until we are happy with the quality of the model.

This is a form of semisupervised learning.

TIP

You can improve the model accuracy further by reusing the discarded images as negative training examples.

NOTE

For a large set of images that don’t have labels, you might want to use co-occurrence of other defining text as labels; for example, hashtags, emojis, alt-text, etc.

Facebook built a dataset of 3.5 billion images using the hashtags from the text of corresponding posts as weak labels, training them, and eventually fine tuning them on the ImageNet dataset. This model beat the state-of-the-art result by 2% (85% top 1% accuracy).

Now that we have collected our image datasets, let’s finally begin training them.

TIP

You can improve the model accuracy further by reusing the discarded images as negative training examples.

NOTE

For a large set of images that don't have labels, you might want to use co-occurrence of other defining text as labels; for example, hashtags, emojis, alt-text, etc.

Facebook built a dataset of 3.5 billion images using the hashtags from the text of corresponding posts as weak labels, training them, and eventually fine tuning them on the ImageNet dataset. This model beat the state-of-the-art result by 2% (85% top 1% accuracy).

Now that we have collected our image datasets, let's finally begin training them.

Training Our Model

Broadly speaking there are three easy ways to train, all of which we have discussed previously. Here, we provide a brief overview of a few different approaches.

Training Our Model

Broadly speaking there are three easy ways to train, all of which we have discussed previously. Here, we provide a brief overview of a few different approaches.

Approach 1: Use Web UI-based Tools

As discussed in Chapter 8, there are several tools to build custom models by supplying labeled images and performing training using the web UI. Microsoft's CustomVision.ai, Google AutoML, IBM Watson Visual Recognition, Clarifai, and Baidu EZDL are a few examples. These methods are code free and many provide simple drag-and-drop GUIs for training.

Let's look at how we can have a mobile-friendly model ready in less than five minutes using CustomVision.ai:

1. Go to *http://customvision.ai*, and make a new project. Because we want to export the trained model to a mobile phone, select a compact model type. Since our domain is food related, select "Food (Compact)," as shown in Figure 12-5.

Create new project

Name*

Not Hotdog Project

Description

Next multi-million dollar project idea!

Resource Group create new

Book [F0]

Manage Resource Group Permissions

Project Types

- (●) Classification
- ( ) Object Detection

Classification Types

- ( ) Multilabel (Multiple tags per image)
- (●) Multiclass (Single tag per image)

Domains:

- ( ) General
- ( ) Food
- ( ) Landmarks
- ( ) Retail
- ( ) General (compact)
- (●) Food (compact)
- ( ) Landmarks (compact)
- ( ) Retail (compact)

Export Capabilities:

- (●) Basic platforms (Tensorflow, CoreML, ONNX, ...)
- ( ) Vision AI Dev Kit

Cancel | Create project

Figure 12-5. Define a new project on CustomVision.ai

2. Upload the images and assign tags (labels), as depicted in Figure 12-6. Upload at least 30 images per tag.

Figure 12-6. Uploading images on the CustomVision.ai dashboard. Note that the tags have been populated as Hotdog and Not Hotdog

3. Click the Train button. A dialog box opens, as shown in Figure 12-7. Fast Training essentially trains the last few layers, whereas Advanced Training can potentially tune the full network giving even higher accuracy (and obviously take more time and money). The Fast Training option should be sufficient for most cases.

Figure 12-7. Options for training type

4. In under a minute, a screen should appear, showing the precision and recall for the newly trained model per category, as shown in Figure 12-8. (This should ring a bell because we had discussed precision and recall earlier in the book.)

Figure 12-8. Precision, Recall, and Average Precision of the newly trained model

5. Play with the probability threshold to see how it changes the model's performance. The default 90% threshold achieves pretty good results. The higher the threshold, the more precise the model becomes, but at the expense of reduced recall.

6. Press the Export button and select the iOS platform (Figure 12-9). Internally, CustomVision.ai converts the model to Core ML (or TensorFlow Lite if you're exporting for Android).

Figure 12-9. The model exporter options in CustomVision.ai

And we're done, all without writing a single line of code! Now let's look at an even more convenient way of training without coding.

Figure 12-9. The model exporter options in CustomVision.ai

And we're done, all without writing a single line of code! Now let's look at an even more convenient way of training without coding.

Approach 2: Use Create ML

In 2018, Apple launched Create ML as a way for developers within the Apple ecosystem to train computer-vision models natively. Developers could open a *playground* and write a few lines of Swift code to train an image classifier. Alternatively, they could use `CreateMLUI` import to display a limited GUI training experience within the playground. It was a good way to get Swift developers to deploy Core ML models without requiring much experience in machine learning.

A year later, at the Apple Worldwide Developers Conference (WWDC) 2019, Apple lowered the barrier even further by announcing the standalone Create ML app on macOS Catalina (10.15). It provides an easy-to-use GUI to train neural networks without needing to write any code at all. Training a neural network simply became a matter of dragging and dropping files into this UI. In addition to supporting the image classifier, they also announced support for object detectors, NLP, sound classification, activity classification (classify activities based on motion sensor data from Apple Watch and iPhone), as well as tabular data (including recommendation systems).

And, it’s fast! Models can be trained in under a minute. This is because it uses transfer learning, so it doesn’t need to train all of the layers in the network. It also supports various data augmentations such as rotations, blur, noise, and so on, and all you need to do is click checkboxes.

Before Create ML came along, it was generally a given that anyone seeking to train a serious neural network in a reasonable amount of time had to own an NVIDIA GPU. Create ML took advantage of the onboard Intel and/or Radeon graphics cards, which allowed faster training on MacBooks without the need to purchase additional hardware. Create ML allows us to train multiple models, from different data sources, all at the same time. It can benefit particularly from powerful hardware such as the Mac Pro or even an external GPU (eGPU).

One major motivation to use Create ML is the size of the models it outputs. A full model can be broken down into a base model (which emits features) and lighter task-specific classification layers. Apple ships the base models into each of its operating systems. So, Create ML just needs to output the task-specific

classifier. How small are these models? As little as a few kilobytes (compared to more than 15 MB for a MobileNet model, which is already pretty small for a CNN). This is important in a day and age when more and more app developers are beginning to incorporate deep learning into their apps. The same neural networks do not need to be unnecessarily replicated across several apps consuming valuable storage space.

In short, Create ML is easy, speedy, and tiny. Sounds too good to be true. Turns out the flip-side of having full vertical integration is that the developers are tied into the Apple ecosystem. Create ML exports only *.mlmodel* files, which can be used exclusively on Apple operating systems such as iOS, iPadOS, macOS, tvOS, and watchOS. Sadly, Android integration is not yet a reality for Create ML.

In this section, we build the Not Hotdog classifier using Create ML:

1. Open the Create ML app, click New Document, and select the Image Classifier template from among the several options available (including Sound, Activity, Text, Tabular), as shown in Figure 12-10. Note that this is only available on Xcode 11 (or greater), on macOS 10.15 (or greater).

Figure 12-10. Choosing a template for a new project

2. In the next screen, enter a name for the project, and then select Done.

3. We need to sort the data into the correct directory structure. As Figure 12-11 illustrates, we place images in directories that have the names of their labels. It is useful to have separate train and test datasets with their corresponding directories.

Figure 12-11. Train and test data in separate directories

4. Point the UI to the training and test data directories, as shown in Figure 12-12.

Figure 12-12. Training interface in Create ML

5. Figure 12-12 shows the UI after you select the train and test data directories. Notice that the validation data was automatically selected by Create ML. Additionally, notice the augmentation options available. It is at this point that we can click the Play button (the right-facing triangle; see Figure 12-13) to start the training process.

Figure 12-13. Create ML screen that opens after loading train and test data

> **NOTE**
>
> As you experiment, you will quickly notice that each augmentation that we add will make the training slower. To set a quick baseline performance metric, we should avoid using augmentations in the first run. Subsequently, we can experiment with adding more and more augmentations to assess how they affect the quality of the model.

6. When the training completes, we can see how the model performed on the training data, (auto-selected) validation data, and the test data, as depicted in Figure 12-14. At the bottom of the screen, we can also observe how long the training process took and the size of the final model. 97% test accuracy in under two minutes. And all that with a 17 KB output. Not too shabby.

Figure 12-14. The Create ML screen after training completes

7. We're so close now—we just need to export the final model. Drag the Output button (highlighted in Figure 12-14) to the desktop to create the *.mlmodel* file.

8. We can double-click on the newly exported *.mlmodel* file to inspect the input and output layers, as well as test drive the model by dropping images into it, as shown in Figure 12-15.

Figure 12-15. The model inspector UI within Xcode

The model is now ready to be plugged into apps on any Apple device.

> **NOTE**
>
> Create ML uses transfer learning, training only the last few layers. Depending on your use case, the underlying model that Apple provides you might be insufficient to make high-quality predictions. This is because you are unable to train the earlier layers in the model, thereby restricting the potential to which the model can be tuned. For most day-to-day problems, this should not be an issue. However, for very domain-specific applications like X-rays, or very similar-looking objects for which the tiniest of details matter (like distinguishing currency notes), training a full CNN would be a better approach. We look at doing so in the following section.

Figure 12-15. The model inspector UI within Xcode

The model is now ready to be plugged into apps on any Apple device.

> **NOTE**
>
> Create ML uses transfer learning, training only the last few layers. Depending on your use case, the underlying model that Apple provides you might be insufficient to make high-quality predictions. This is because you are unable to train the earlier layers in the model, thereby restricting the potential to which the model can be tuned. For most day-to-day problems, this should not be an issue. However, for very domain-specific applications like X-rays, or very similar-looking objects for which the tiniest of details matter (like distinguishing currency notes), training a full CNN would be a better approach. We look at doing so in the following section.

Approach 3: Fine Tuning Using Keras

By now, we have become experts in using Keras. This option can get us even higher accuracy if we are up for experimentation and are willing to spend more time training the model. Let's reuse the code from Chapter 3 and modify parameters such as directory and filename, batch size, and number of images. You will find the code on the book's GitHub website (see *http://PracticalDeepLearning.ai*) at *code/chapter-12/1-keras-custom-classifier-with-transfer-learning.ipynb*.

The model training should take a few minutes to complete, depending on the hardware, and at the end of the training, we should have a *NotHotDog.h5* file ready on the disk.

Approach 3: Fine Tuning Using Keras

By now, we have become experts in using Keras. This option can get us even higher accuracy if we are up for experimentation and are willing to spend more time training the model. Let's reuse the code from Chapter 3 and modify parameters such as directory and filename, batch size, and number of images. You will find the code on the book's GitHub website (see *http://PracticalDeepLearning.ai*) at *code/chapter-12/1-keras-custom-classifier-with-transfer-learning.ipynb*.

The model training should take a few minutes to complete, depending on the hardware, and at the end of the training, we should have a *NotHotDog.h5* file ready on the disk.

Model Conversion Using Core ML Tools

As discussed in Chapter 11, there are several ways of converting our models to the Core ML format.

Models generated from CustomVision.ai are directly available in Core ML format, hence no conversion is necessary. For models trained in Keras, Core ML Tools can help convert as follows. Note that because we are using a MobileNet model, which uses a custom layer called `relu6`, we need to import `CustomObjectScope`:

```
from tensorflow.keras.models import load_model
from tensorflow.keras.utils.generic_utils import CustomObjectScope
import tensorflow.keras

with CustomObjectScope({'relu6':
tensorflow.keras.applications.mobilenet.relu6,'DepthwiseConv2D':
tensorflow.keras.applications.mobilenet.DepthwiseConv2D}):
    model = load_model('NotHotDog-model.h5')

import coremltools
coreml_model = coremltools.converters.keras.convert(model)
coreml_model.save('NotHotDog.mlmodel')
```

Now that we have a Core ML model ready, all we need to do is build the app.

Model Conversion Using Core ML Tools

As discussed in Chapter 11, there are several ways of converting our models to the Core ML format.

Models generated from CustomVision.ai are directly available in Core ML format, hence no conversion is necessary. For models trained in Keras, Core ML Tools can help convert as follows. Note that because we are using a MobileNet model, which uses a custom layer called `relu6`, we need to import `CustomObjectScope`:

```
from tensorflow.keras.models import load_model
from tensorflow.keras.utils.generic_utils import CustomObjectScope
import tensorflow.keras

with CustomObjectScope({'relu6':
tensorflow.keras.applications.mobilenet.relu6,'DepthwiseConv2D':
tensorflow.keras.applications.mobilenet.DepthwiseConv2D}):
    model = load_model('NotHotDog-model.h5')

import coremltools
coreml_model = coremltools.converters.keras.convert(model)
coreml_model.save('NotHotDog.mlmodel')
```

Now that we have a Core ML model ready, all we need to do is build the app.

Building the iOS App

We can use the code from Chapter 11 and simply replace the *.mlmodel* with the newly generated model file, as demonstrated in Figure 12-16.

Figure 12-16. Loading the `.mlmodel` *into Xcode*

Now, compile and run the app and you're done! Figure 12-17 presents the awesome results.

Figure 12-17. Our app identifying the hot dog

Further Exploration

Can we make this application more interesting? We can build an actual “Shazam for food” by training for all the categories in the Food-101 dataset, which we cover in the next chapter. Additionally, we can enhance the UI compared to the barebones percentages our current app shows. And, to make it viral just like “Not Hotdog,” provide a way to share the classifications to social media platforms.

Further Exploration

Can we make this application more interesting? We can build an actual “Shazam for food” by training for all the categories in the Food-101 dataset, which we cover in the next chapter. Additionally, we can enhance the UI compared to the barebones percentages our current app shows. And, to make it viral just like “Not Hotdog,” provide a way to share the classifications to social media platforms.

Summary

In this chapter, we worked through an end-to-end pipeline of collecting data, training and converting a model, and using it in the real world on an iOS device. For each step of the pipeline, we have explored a few different options in varying degrees of complexity. And, we have placed the concepts covered in the previous chapters in the context of a real-world application.

And now, like Jian-Yang, go make your millions!

Summary

In this chapter, we worked through an end-to-end pipeline of collecting data, training and converting a model, and using it in the real world on an iOS device. For each step of the pipeline, we have explored a few different options in varying degrees of complexity. And, we have placed the concepts covered in the previous chapters in the context of a real-world application.

And now, like Jian-Yang, go make your millions!

Chapter 13. Shazam for Food: Developing Android Apps with TensorFlow Lite and ML Kit

After developing the viral Not Hotdog app (that we looked at in Chapter 12), Jian-Yang was originally supposed to build a classifier to recognize all food in existence. In fact, the app was originally supposed to be called SeeFood—an app that can "see" food and know it right away (Figure 13-1). In other words, the "Shazam for Food." However, the app was too successful for its own good and was acquired by Periscope. The original vision of his investor, Erlich Bachman, remains unfulfilled. In this chapter, our mission is to fulfill this dream.

Figure 13-1. Not Hotdog app listing on the Apple App Store

Where would such a feature be useful? For health nuts, it could look at a dish and provide the nutritional information, including the number of calories. Or, it could scan a few ingredients, and recommend a recipe based on them. Or, it could even look at a product in the market, and check whether it contains any blacklisted ingredients such as specific allergens.

This is an interesting problem to solve for several reasons because it represents several challenges:

Data collection challenge

There are more than a hundred cuisines around the world, each with hundreds if not thousands of dishes.

Accuracy challenge

It should be right most of the time.

Performance challenge

It should run near instantly.

Platform challenge

An iPhone app alone would be insufficient. A lot of users in developing countries use less powerful smartphones, particularly Android devices. Cross-platform development is a must.

Making a food classifier app for one cuisine is tricky enough. Imagine having to do that for every food in existence—and doing it on two platforms! An individual or a small team will quickly run into scaling issues trying to tackle this problem. In this chapter, we use this example as a motivation to explore the different parts of the mobile AI development life cycle that we explored in Chapter 11.

The material we explore here does not need to be limited to smartphones, either. We can apply our learnings beyond mobile to edge devices such as Google Coral and Raspberry Pi, which we discuss later in the book.

Data collection challenge

There are more than a hundred cuisines around the world, each with hundreds if not thousands of dishes.

Accuracy challenge

It should be right most of the time.

Performance challenge

It should run near instantly.

Platform challenge

An iPhone app alone would be insufficient. A lot of users in developing countries use less powerful smartphones, particularly Android devices. Cross-platform development is a must.

Making a food classifier app for one cuisine is tricky enough. Imagine having to do that for every food in existence—and doing it on two platforms! An individual or a small team will quickly run into scaling issues trying to tackle this problem. In this chapter, we use this example as a motivation to explore the different parts of the mobile AI development life cycle that we explored in Chapter 11.

The material we explore here does not need to be limited to smartphones, either. We can apply our learnings beyond mobile to edge devices such as Google Coral and Raspberry Pi, which we discuss later in the book.

The Life Cycle of a Food Classifier App

So, we want to build a global multicuisine, multiplatform food classifier. It sounds like a daunting task, but we can break it down into manageable steps. As in life, we first need to crawl, then walk, and then run. The following is one potential approach to consider:

1. Collect a small initial set images for a single cuisine (e.g., Italian).
2. Label these images with their corresponding dish identifiers (e.g., `margherita_pizza`).
3. Train a classifier model.
4. Convert the model to a mobile framework-compatible format (e.g., *.tflite*).
5. Build a mobile app by integrating the model with a great UX.
6. Recruit alpha users and share the app with them.
7. Collect detailed usage metrics along with feedback from active users, including camera frames (which tend to reflect real-world usage) and corresponding proxy labels (indicating whether the classification was right or wrong).
8. Improve the model using the newly collected images as additional training data. This process needs to be iterative.
9. When the quality of the model meets the minimum quality bar, ship the app/feature to more/all users. Continue to monitor and improve the quality of the model for that cuisine.
10. Repeat these steps for each cuisine.

TIP

For step 7, we could alternatively integrate the feedback within the UX itself. For example, the app could show a ranked list of predictions (by probability) that are likely candidates for a given picture. If our model is doing well, the user should be selecting the first option most of the time. Selecting a lower ranked prediction would essentially be considered an incorrect prediction. In the worst case, if none of

the options were correct, allow the user to manually add a new label. That photo, along with the label (in all three scenarios), can be incorporated as training data.

We don’t need a whole lot of data to get underway. Although each of the aforementioned steps might sound involved, we can significantly automate the process. What’s cool about this approach is that the more the app is used, the better it becomes, automatically. It’s as if it has a life of its own. We explore this self-evolving approach for a model toward the end of the chapter.

TIP

A die-hard fan of your app/company would make for a good alpha user. Alpha users ideally are people who have a vested interest in the success of your product. For a food recognition app, a potential group of users could be fitness buffs who watch every calorie and ingredient they eat. These users understand that the quality of the app will not be up to par early on, but they also see their role in shaping it through consistent, constructive feedback. They voluntarily agree to a liberal data-sharing agreement in order to provide data such as usage metrics and image frames from day-to-day use. We would recommend that your users know exactly what information you would be collecting about them and allow them to opt out or delete. Don’t be creepy!

In this chapter, we explore the different parts of the aforementioned life cycle and the tools that help us with each of those steps. In the end, we will take a holistic, end-to-end look at the entire mobile development life cycle, not just from this chapter, but also from the previous chapters, and combine them to see how we could effectively use them in building a production-quality, real-world application.

Our journey begins with understanding the following tools from the Google ecosystem.

TensorFlow Lite

Model conversion and mobile inference engine.

ML Kit

High-level software development kit (SDK) with several built-in APIs, along with the ability to run custom TensorFlow Lite models as well as integration with Firebase on Google Cloud.

Firebase

A cloud-based framework that provides the necessary infrastructure for production-quality mobile applications, including analytics, crash reporting, A/B testing, push notifications, and more.

TensorFlow Model Optimization Toolkit

A set of tools for optimizing the size and performance of models.

Firebase

A cloud-based framework that provides the necessary infrastructure for production-quality mobile applications, including analytics, crash reporting, A/B testing, push notifications, and more.

TensorFlow Model Optimization Toolkit

A set of tools for optimizing the size and performance of models.

An Overview of TensorFlow Lite

As mentioned in Chapter 11, Google released an on-device inference engine called TensorFlow Lite that expanded the reach of the TensorFlow ecosystem beyond the cloud and desktops. Prior to this, the options within the TensorFlow ecosystem were porting the entire TensorFlow library itself to iOS (which was heavy and slow) and, later on, its slightly stripped-down version called TensorFlow Mobile (an improvement, but still fairly bulky).

TensorFlow Lite is optimized from the ground up for mobile, with the following salient features:

Small

TensorFlow Lite comes packaged with a much lighter interpreter. Even with all the operators included, the interpreter is less than 300 KB. In typical usage with common models like MobileNet, we can expect that footprint to be less than 200 KB. For reference, the previous generation TensorFlow Mobile used to occupy 1.5 MB. Additionally, TensorFlow Lite uses s*elective registration*—it packages only the operations that it knows will be used by the model, minimizing unnecessary overheads.

Fast

TensorFlow Lite provides a significant speedup because it is able to take advantage of on-device *hardware acceleration such as* GPUs and NPUs, where available. In the Android ecosystem, it uses the Android Neural Networks API for acceleration. Analogously on iPhones, it uses the Metal API. Google claims two to seven times speedup over a range of tasks when using GPUs (relative to CPUs).

TensorFlow uses Protocol Buffers (Protobufs) for deserialization/serialization. Protobufs are a powerful tool for representing data due to their flexibility and extensibility. However, that comes at a performance cost that can be felt on low-power devices such as mobiles.

FlatBuffers turned out to be the answer to this problem. Originally built for video game development, for which low overhead and high performance are

a must, they proved to be a good solution for mobiles as well in significantly reducing the code footprint, memory usage, and CPU cycles spent for serialization and deserialization of models. This also improved start-up time by a fair amount.

Within a network, there are some layers that have fixed computations at inference time; for example, the batch normalization layers, which can be precomputed because they rely on values obtained during training, such as mean and standard deviation. So, the batch normalization layer computation can be fused (i.e., combined) with the previous layer's computation ahead of time (i.e., during model conversion), thereby reducing the inference time and making the entire model much faster. This is known as *prefused activation*, which TensorFlow Lite supports.

The interpreter uses static memory and a static execution plan. This helps in decreasing the model load time.

Fewer dependencies

The TensorFlow Lite codebase is mostly standard C/C++ with a minimal number of dependencies. It makes it easier to package and deploy and additionally reduces the size of the deployed package.

Supports custom operators

TensorFlow Lite contains quantized and floating-point core operators, many of which have been tuned for mobile platforms, and can be used to create and run custom models. If TensorFlow Lite does not support an operation in our model, we can also write custom operators to get our model running.

Before we build our initial Android app, it would be useful to examine TensorFlow Lite's architecture.

FROM THE CREATOR'S DESK

By Pete Warden, technical lead for Mobile and Embedded TensorFlow, author of *TinyML* (O'Reilly)

When I wrote my first neural network inference engine back in 2013, it was because I couldn't find any other way to run image recognition models on

the cheap AWS servers that were all we could afford at our startup. It was a happy accident that the same code was easy to build for iOS and Android, and I quickly realized that the possibilities for deep learning on mobile platforms were endless. That code became the Jetpac SDK, and I had tremendous fun seeing all the applications people built with it in the early days. When we were acquired by Google, I met Yangqing Jia (of Caffe fame), who was working on an internal 20% project to run inference on mobile devices, so we ended up collaborating and eventually a lot of what we learned ended up in the public release of TensorFlow. A lot of people were skeptical about the value of neural networks on phones, but Jeff Dean was always a firm supporter and he helped us ensure we had Android support right from the start, and iOS very soon after. We learned a lot from those initial TensorFlow releases, and the Mobile Vision team within Google had done some amazing work with their own specialized internal framework, so we joined forces to create TensorFlow Lite as a library offering TensorFlow compatibility, but with a much smaller footprint.

Since launch, TensorFlow Lite has been shipped in production on over two billion devices, but none of that could have happened without the community that’s grown up around it. It’s been amazing to see all the contributions, examples, and peer-to-peer support that has enabled people to build applications that wouldn’t have been possible without the ecosystem working together.

the cheap AWS servers that were all we could afford at our startup. It was a happy accident that the same code was easy to build for iOS and Android, and I quickly realized that the possibilities for deep learning on mobile platforms were endless. That code became the Jetpac SDK, and I had tremendous fun seeing all the applications people built with it in the early days. When we were acquired by Google, I met Yangqing Jia (of Caffe fame), who was working on an internal 20% project to run inference on mobile devices, so we ended up collaborating and eventually a lot of what we learned ended up in the public release of TensorFlow. A lot of people were skeptical about the value of neural networks on phones, but Jeff Dean was always a firm supporter and he helped us ensure we had Android support right from the start, and iOS very soon after. We learned a lot from those initial TensorFlow releases, and the Mobile Vision team within Google had done some amazing work with their own specialized internal framework, so we joined forces to create TensorFlow Lite as a library offering TensorFlow compatibility, but with a much smaller footprint.

Since launch, TensorFlow Lite has been shipped in production on over two billion devices, but none of that could have happened without the community that’s grown up around it. It’s been amazing to see all the contributions, examples, and peer-to-peer support that has enabled people to build applications that wouldn’t have been possible without the ecosystem working together.

TensorFlow Lite Architecture

Figure 13-2 provides a high-level view of the TensorFlow Lite architecture.

Figure 13-2. High-level architecture of the TensorFlow Lite ecosystem

As app developers, we will be working in the topmost layer while interacting with the TensorFlow Lite API (or optionally with ML Kit, which in turn uses TensorFlow Lite). The TensorFlow Lite API abstracts away all of the complexities involved in using a lower-level API such as the Android's Neural Network API. Recall that this is similar to how Core ML works within the Apple ecosystem.

Looking at the other extreme, computations can be run on various types of hardware modules. The most common among them is the CPU, simply because of its ubiquity and flexibility. Modern smartphones are increasingly equipped with specialized modules, including the GPU and the newer NPU (especially built for neural network computations like on the iPhone X). Additionally, Digital Signal Processors (DSP) specialize in singular tasks such as facial authentication, fingerprint authentication, and wake word detection (like "Hey

Siri”).

In the world of Internet of Things (IoT), microcontrollers (MCUs) reign supreme. With no OS, no processor, and very little memory (KBs), these are cheap to produce in mass quantities and easy to incorporate into various applications. With TensorFlow Lite for Microcontrollers, developers can run AI on these bare-metal devices without needing internet connectivity. The pared-down version (roughly 20 KB) of the TensorFlow Lite Interpreter for MCUs is called the TensorFlow Lite Micro Interpreter.

So how does TensorFlow Lite interact with hardware? By using delegates that are platform-aware objects that expose a consistent platform-agnostic API. In other words, delegates shield the interpreter from needing to know anything about the specific hardware it runs on. They take on all or part of the responsibility of graph execution that would have otherwise been run on the CPU and instead run on the much more efficient GPUs and NPUs. On Android, the GPU delegate accelerates performance using OpenGL, whereas on iOS, the Metal API is used.

Given that TensorFlow Lite by itself is platform agnostic, it needs to call into a platform-specific library that implements a known contract. This contract is the TensorFlow Lite Delegate API. Within Android, this contract is fulfilled by the Android Neural Network API (available on devices running Android 8.1 and above). The Neural Network API is designed to provide a base layer of functionality for higher-level machine learning frameworks. The equivalent of the Neural Network API within the Apple world is Metal Performance Shaders.

With the information we have looked at so far, let’s get hands on.

Siri”).

In the world of Internet of Things (IoT), microcontrollers (MCUs) reign supreme. With no OS, no processor, and very little memory (KBs), these are cheap to produce in mass quantities and easy to incorporate into various applications. With TensorFlow Lite for Microcontrollers, developers can run AI on these bare-metal devices without needing internet connectivity. The pared-down version (roughly 20 KB) of the TensorFlow Lite Interpreter for MCUs is called the TensorFlow Lite Micro Interpreter.

So how does TensorFlow Lite interact with hardware? By using delegates that are platform-aware objects that expose a consistent platform-agnostic API. In other words, delegates shield the interpreter from needing to know anything about the specific hardware it runs on. They take on all or part of the responsibility of graph execution that would have otherwise been run on the CPU and instead run on the much more efficient GPUs and NPUs. On Android, the GPU delegate accelerates performance using OpenGL, whereas on iOS, the Metal API is used.

Given that TensorFlow Lite by itself is platform agnostic, it needs to call into a platform-specific library that implements a known contract. This contract is the TensorFlow Lite Delegate API. Within Android, this contract is fulfilled by the Android Neural Network API (available on devices running Android 8.1 and above). The Neural Network API is designed to provide a base layer of functionality for higher-level machine learning frameworks. The equivalent of the Neural Network API within the Apple world is Metal Performance Shaders.

With the information we have looked at so far, let’s get hands on.

Model Conversion to TensorFlow Lite

At this point in the book, we should already have a model at hand (either pretrained on ImageNet or custom trained in Keras). Before we can plug that model into an Android app, we need to convert it to the TensorFlow Lite format (a *.tflite* file).

Let's take a look at how to convert the model using the TensorFlow Lite Converter tool, the `tflite_convert` command that comes bundled with our TensorFlow installation:

```
# Keras to TensorFlow Lite
$ tflite_convert \
  --output_file=my_model.tflite \
  --keras_model_file=my_model.h5

# TensorFlow to TensorFlow Lite
$ tflite_convert \
  --output_file=my_model.tflite \
  --graph_def_file=my_model/frozen_graph.pb
```

The output of this command is the new *my_model.tflite* file, which we can then plug into the Android app in the following section. Later, we look at how to make that model more performant by using the `tflite_convert` tool again. Additionally, the TensorFlow Lite team has created many pretrained models that are available in TensorFlow Lite format, saving us this conversation step.

Model Conversion to TensorFlow Lite

At this point in the book, we should already have a model at hand (either pretrained on ImageNet or custom trained in Keras). Before we can plug that model into an Android app, we need to convert it to the TensorFlow Lite format (a *.tflite* file).

Let's take a look at how to convert the model using the TensorFlow Lite Converter tool, the `tflite_convert` command that comes bundled with our TensorFlow installation:

```
# Keras to TensorFlow Lite
$ tflite_convert \
  --output_file=my_model.tflite \
  --keras_model_file=my_model.h5

# TensorFlow to TensorFlow Lite
$ tflite_convert \
  --output_file=my_model.tflite \
  --graph_def_file=my_model/frozen_graph.pb
```

The output of this command is the new *my_model.tflite* file, which we can then plug into the Android app in the following section. Later, we look at how to make that model more performant by using the `tflite_convert` tool again. Additionally, the TensorFlow Lite team has created many pretrained models that are available in TensorFlow Lite format, saving us this conversation step.

Building a Real-Time Object Recognition App

Running the sample app from the TensorFlow repository is an easy way to play with the TensorFlow Lite API. Note that we would need an Android phone or tablet to run the app. Following are steps to build and deploy the app:

1. Clone the TensorFlow repository:

```
git clone https://github.com/tensorflow/tensorflow.git
```

2. Download and install Android Studio from *https://developer.android.com/studio*.

3. Open Android Studio and then select "Open an existing Android Studio project" (Figure 13-3).

Figure 13-3. Start screen of Android Studio

4. Go to the location of the cloned TensorFlow repository and then navigate further to *tensorflow/tensorflow/contrib/lite/java/demo/* (Figure 13-4). Select Open.

Figure 13-4. Android Studio "Open Existing Project" screen in the TensorFlow repository

5. On the Android device, enable Developer Options. (Note that we used a Pixel device here, which uses the stock Android OS. For other manufacturers, the instructions might be a little different.)

   a. Go to Settings.

   b. Scroll down to the About Phone or About Tablet option (Figure 13-5) and select it.

11:12
System
About phone
Pixel
Languages & input
Gboard
Gestures
Date & time
GMT-07:00 Pacific Daylight Time
Backup
Off
Reset options
Network, apps, or device can be reset
Multiple users
Signed in as Owner
System update
Updated to Android 9

Figure 13-5. System information screen on an Android phone; select the About Phone option here

c. Look for the Build Number row and tap it seven times. (Yeah, you read that right—seven!)

d. You should see a message (Figure 13-6) confirming that developer mode is enabled.

Figure 13-6. The About Phone screen on an Android device

e. If you are using a phone, tap the back button to go back to the previous menu.

f. You should see a “Developer options” button, directly above the “About phone” or “About tablet” option (Figure 13-7). Tab this button to reveal the “Developer options” menu (Figure 13-8).

s

11:13
System
About phone
Pixel
Languages & input
Gboard
Gestures
Date & time
GMT-07:00 Pacific Daylight Time
Backup
Off
Reset options
Network, apps, or device can be reset
Multiple users
Signed in as Owner
Developer options
System update
Updated to Android 9

Figure 13-7. The System information screen showing “Developer options” enabled

11:13
Developer options
On
System UI demo mode
Night mode
Always off
Quick settings developer tiles
Debugging
USB debugging
Debug mode when USB is connected
Revoke USB debugging authorizations
Bug report shortcut
Show a button in the power menu for taking a bug report
Select mock location app
No mock location app set
Force full GNSS measurements

Figure 13-8. “Developer options” screen on an Android device with USB debugging enabled

6. Plug the Android device into the computer via a USB cable.
7. The Android device might show a message asking to allow USB debugging. Enable “Always allow this computer,” and then select OK (Figure 13-9).

11:14
Sunday, Aug 12
Settings
Allow USB debugging?
The computer's RSA key fingerprint is:
16:B4:91:9C:E0:C4:E1:E9:C4:1A:C3:FF:6A:
98:DD:61
Always allow from this computer
Cancel
OK
Maps
Play Store

Figure 13-9. Allow USB debugging on the displayed alert

8. In Android Studio, on the Debug toolbar bar (Figure 13-10), click the Run App button (the right-facing triangle).

Figure 13-10. Debug toolbar in Android Studio

9. A window opens displaying all the available devices and emulators (Figure 13-11). Choose your device, and then select OK.

Figure 13-11. Select the phone from the deployment target selection screen

10. The app should install and begin running on our phone.
11. The app will request permission for your camera; go ahead and grant it permission.
12. A live view of the camera should appear, along with real-time predictions o object classification, plus the number of seconds it took to make the prediction, as shown in Figure 13-12.

f

8:07
CameraApp
230 ms 66 ms [[882] upright, upright piano (48.2%), [883] vacuum, vacuum cleaner (47.5%), [743] printer (47.1%), [778] scabbard (47.1%), [765] rifle (46.7%), [402] accordion, piano accordion, squeeze box (45.9%), [584] guillotine (45.9%), [688] organ, pipe organ (45.9%), [770] rule, ruler (45.5%), [580] grand piano, grand (45.1%)]

Figure 13-12. The app up-and-running app, showing real-time predictions

And there you have it! We have a basic app running on the phone that takes video frames and classifies them. It's simple and it works reasonably well.

Beyond object classification, the TensorFlow Lite repository also has sample apps (iOS and Android) for many other AI problems, including the following:

- Object detection
- Pose estimation
- Gesture recognition
- Speech recognition

The great thing about having these sample apps is that with basic instructions, someone without a mobile development background can get them running on a phone. Even better, if we have a custom trained model, we can plug it into the app and see it run for our custom task.

This is great for starting out. However, things are a lot more complicated in the real world. Developers of serious real-world applications with thousands or even millions of users need to think beyond just inference—like updating and distributing models, testing different versions among subsets of users, maintaining parity between iOS and Android, and ultimately reducing the engineering cost of each. Doing all of this in-house can be expensive, time consuming, and frankly unnecessary. Naturally, platforms that provide these features would be enticing. This is where ML Kit and Firebase come in.

Figure 13-12. The app up-and-running app, showing real-time predictions

And there you have it! We have a basic app running on the phone that takes video frames and classifies them. It's simple and it works reasonably well.

Beyond object classification, the TensorFlow Lite repository also has sample apps (iOS and Android) for many other AI problems, including the following:

- Object detection
- Pose estimation
- Gesture recognition
- Speech recognition

The great thing about having these sample apps is that with basic instructions, someone without a mobile development background can get them running on a phone. Even better, if we have a custom trained model, we can plug it into the app and see it run for our custom task.

This is great for starting out. However, things are a lot more complicated in the real world. Developers of serious real-world applications with thousands or even millions of users need to think beyond just inference—like updating and distributing models, testing different versions among subsets of users, maintaining parity between iOS and Android, and ultimately reducing the engineering cost of each. Doing all of this in-house can be expensive, time consuming, and frankly unnecessary. Naturally, platforms that provide these features would be enticing. This is where ML Kit and Firebase come in.

ML Kit + Firebase

ML Kit is a mobile SDK launched during the 2018 Google I/O conference. It provides convenient APIs for novices as well as advanced ML developers to do a lot of common ML tasks. By default, ML Kit comes with a generic feature set in vision and language intelligence. Table 4-1 lists some of the common ML tasks that we can do in just a few lines.

Table 13-1. ML Kit built-in features

Vision	Language
Object classification	Language identification
Object detection and tracking	On-device translation
Popular landmark detection	Smart replies
Text recognition	
Face detection	
Barcode detection	

ML Kit also gives us the ability to use custom trained TensorFlow Lite models for inference. Let's take a moment to appreciate why this is so great for developers. Imagine that we were building a business card scanner. We could bring in a custom business card detection model, recognize when a business card is visible and its boundaries (to make a nice visual user interface), run the built-in text recognition, and filter out text outside of those boundaries to prevent extraneous characters. Or, consider a language learning game that can be built by pointing at objects, running an object classifier, and then using the on-device translation API to announce the labels in French. It is entirely possible to build these relatively quickly using ML Kit. Although many of these features are available in Core ML, too, ML Kit has the added advantage of being cross-platform.

ML Kit, though, is only one piece of the puzzle. It integrates into Google's Firebase, the mobile and web application development platform that is part of Google Cloud. Firebase offers an array of features that are necessary

infrastructure for production-quality apps, such as the following:

- Push notifications
- Authentication
- Crash reporting
- Logging
- Performance monitoring
- Hosted device testing
- A/B testing
- Model management

The last point is very pertinent to us. One of the biggest benefits that Firebase gives us is the ability to host our custom models on the cloud and download them within the app as needed. Simply copy the models over to Firebase on Google Cloud, reference the model on ML Kit inside the app, and we're good to go. The A/B testing feature gives us the ability to show different users different versions of the same model and measure the performance across the different models.

NOTE

For a lot of the built-in functionality, ML Kit optionally provides a feature to process images in the cloud, where the models are much larger compared to on-device models. These larger models obviously require more powerful hardware to run them, with the benefit of providing some improvement in accuracy and potentially a much larger taxonomy (like thousands of object classes instead of hundreds). In fact, some functionality such as the landmark recognition feature works only on the cloud.

The cloud processing option is particularly useful when we need a little bit of extra accuracy and/or the user's phone has low processing power that prevents it from running the on-device model well.

infrastructure for production-quality apps, such as the following:

- Push notifications
- Authentication
- Crash reporting
- Logging
- Performance monitoring
- Hosted device testing
- A/B testing
- Model management

The last point is very pertinent to us. One of the biggest benefits that Firebase gives us is the ability to host our custom models on the cloud and download them within the app as needed. Simply copy the models over to Firebase on Google Cloud, reference the model on ML Kit inside the app, and we're good to go. The A/B testing feature gives us the ability to show different users different versions of the same model and measure the performance across the different models.

NOTE

For a lot of the built-in functionality, ML Kit optionally provides a feature to process images in the cloud, where the models are much larger compared to on-device models. These larger models obviously require more powerful hardware to run them, with the benefit of providing some improvement in accuracy and potentially a much larger taxonomy (like thousands of object classes instead of hundreds). In fact, some functionality such as the landmark recognition feature works only on the cloud.

The cloud processing option is particularly useful when we need a little bit of extra accuracy and/or the user's phone has low processing power that prevents it from running the on-device model well.

Object Classification in ML Kit

For our previous task of object classification in real time, if we use ML Kit instead of vanilla TensorFlow Lite, we can simplify our code to just the following lines (in Kotlin):

```
val image = FirebaseVisionImage.fromBitmap(bitmap)
val detector = FirebaseVision.getInstance().visionLabelDetector
val result = detector.detectInImage(image).addOnSuccessListener { labels ->
    // Print labels
}
```

Object Classification in ML Kit

For our previous task of object classification in real time, if we use ML Kit instead of vanilla TensorFlow Lite, we can simplify our code to just the following lines (in Kotlin):

```
val image = FirebaseVisionImage.fromBitmap(bitmap)
val detector = FirebaseVision.getInstance().visionLabelDetector
val result = detector.detectInImage(image).addOnSuccessListener { labels ->
    // Print labels
}
```

Custom Models in ML Kit

In addition to the prebuilt models provided by ML Kit, we can also run our own custom models. These models must be in the TensorFlow Lite format. Following is a simple piece of code to load a custom model that's bundled into the app:

```
val customModel = FirebaseLocalModelSource.Builder("my_custom_model")
        .setAssetFilePath("my_custom_model.tflite").build()
FirebaseModelManager.getInstance().registerLocalModelSource(customModel)
```

Next, we specify the model's input and output configuration (for a model that takes in an RGB image of size 224x224 and gives predictions for 1,000 class names):

```
val IMAGE_WIDTH = 224
val IMAGE_HEIGHT = 224
val modelConfig = FirebaseModelInputOutputOptions.Builder()
        .setInputFormat(0, FirebaseModelDataType.FLOAT32, intArrayOf(1,
IMAGE_WIDTH, IMAGE_HEIGHT, 3))
        .setOutputFormat(0, FirebaseModelDataType.FLOAT32, intArrayOf(1, 1000))
        .build()
```

Next, we create an array of a single image and normalize each pixel to the range [–1,1]:

```
val bitmap = Bitmap.createScaledBitmap(image, IMAGE_WIDTH, IMAGE_HEIGHT, true)
val input = Array(1) {
        Array(IMAGE_WIDTH) { Array(IMAGE_HEIGHT) { FloatArray(3) } }
        }
for (x in 0..IMAGE_WIDTH) {
    for (y in 0..IMAGE_HEIGHT) {
        val pixel = bitmap.getPixel(x, y)
        input[0][x][y][0] = (Color.red(pixel) - 127) / 128.0f
        input[0][x][y][1] = (Color.green(pixel) - 127) / 128.0f
        input[0][x][y][2] = (Color.blue(pixel) - 127) / 128.0f
    }
}
```

Now, we set up an interpreter based on our custom model:

```
val options = FirebaseModelOptions.Builder()
     .setLocalModelName("my_custom_model").build()
```

```
val interpreter = FirebaseModelInterpreter.getInstance(options)
```

Next, we run our input batch on the interpreter:

```
val modelInputs = FirebaseModelInputs.Builder().add(input).build()
interpreter.run(modelInputs, modelConfig).addOnSuccessListener { result ->
    // Print results
}
```

Yup, it's really that simple! Here, we've seen how we can bundle custom models along with the app. Sometimes, we might want the app to dynamically download the model from the cloud for reasons such as the following:

- We want to keep the default app size small on the Play Store so as to not prevent users with data usage constraints from downloading our app.
- We want to experiment with a different variety of models and pick the best one based on the available metrics.
- We want the user to have the latest and greatest model, without having to go through the whole app release process.
- The feature that needs the model might be optional, and we want to conserve space on the user's device.

This brings us to hosted models.

```
val interpreter = FirebaseModelInterpreter.getInstance(options)
```

Next, we run our input batch on the interpreter:

```
val modelInputs = FirebaseModelInputs.Builder().add(input).build()
interpreter.run(modelInputs, modelConfig).addOnSuccessListener { result ->
    // Print results
}
```

Yup, it's really that simple! Here, we've seen how we can bundle custom models along with the app. Sometimes, we might want the app to dynamically download the model from the cloud for reasons such as the following:

- We want to keep the default app size small on the Play Store so as to not prevent users with data usage constraints from downloading our app.
- We want to experiment with a different variety of models and pick the best one based on the available metrics.
- We want the user to have the latest and greatest model, without having to go through the whole app release process.
- The feature that needs the model might be optional, and we want to conserve space on the user's device.

This brings us to hosted models.

Hosted Models

ML Kit, along with Firebase, gives us the ability to upload and store our model on Google Cloud and download it from the app when needed. After the model is downloaded, it functions exactly like it would have if we had bundled the model into the app. Additionally, it provides us with the ability to push updates to the model without having to do an entire release of the app. Also, it lets us do experiments with our models to see which ones perform best in the real world. For hosted models, there are two aspects that we need to look at.

Accessing a hosted model

The following lines inform Firebase that we'd like to use the model named `my_remote_custom_model:`

```
val remoteModel = FirebaseCloudModelSource.Builder("my_remote_custom_model")
        .enableModelUpdates(true).build()
FirebaseModelManager.getInstance().registerCloudModelSource(remoteModel)
```

Notice that we set `enableModelUpdates` to enable us to push updates to the model from the cloud to the device. We can also optionally configure the conditions under which the model would be downloaded for the first time versus every subsequent time—whether the device is idle, whether it's currently charging, and whether the download is restricted to WiFi networks only.

Next, we set up an interpreter much like we did with our local model:

```
val options = FirebaseModelOptions.Builder()
    .setCloudModelName("my_remote_custom_model").build()
val interpreter = FirebaseModelInterpreter.getInstance(options)
```

After this point, the code to perform the prediction would look exactly the same as that for local models.

Next, we discuss the other aspect to hosted models—uploading the model.

Uploading a hosted model

As of this writing, Firebase supports only models hosted on GCP. In this section,

we walk through the simple process of creating, uploading, and storing a hosted model. This subsection presumes that we already have an existing GCP account.

The following lists the steps that we need to take to get a model hosted on the cloud:

1. Go to *https://console.firebase.google.com*. Select an existing project or add a new one (Figure 13-13).

Figure 13-13. Home page of Google Cloud Firebase

2. On the Project Overview screen, create an Android app (Figure 13-14).

Figure 13-14. The Project Overview screen on Google Cloud Firebase

3. Use the app ID from the project in Android Studio (Figure 13-15).

Figure 13-15. App creation screen on Firebase

4. After clicking “Register app,” download the configuration file. This configuration file gives the necessary credentials to the app for it to access our cloud account. Add the configuration file and the Firebase SDK to the Android app as shown on the app creation page.

5. In the ML Kit section, select Get Started, and then select “Add custom model” (Figure 13-16).

Figure 13-16. The ML Kit custom models tab

6. In the name field, enter `my_remote_custom_model` to match the name in the code.
7. Upload the model file from your computer (Figure 13-17).

Figure 13-17. Uploading a TensorFlow Lite model file to Firebase

8. Tap the “Publish” button after the file upload completes.

That’s it! Our model is now ready to be accessed and used from the app dynamically. Next, we examine how we can do A/B testing between models using Firebase.

Figure 13-17. Uploading a TensorFlow Lite model file to Firebase

8. Tap the “Publish” button after the file upload completes.

That’s it! Our model is now ready to be accessed and used from the app dynamically. Next, we examine how we can do A/B testing between models using Firebase.

A/B Testing Hosted Models

Let's take the scenario in which we had a version 1 model named `my_model_v1` to start off with and deployed it to our users. After some usage by them, we obtained more data that we were able to train on. The result of this training was `my_model_v2` (Figure 13-18). We want to assess whether this new version would give us better results. This is where A/B testing comes in.

Figure 13-18. Currently uploaded custom models to Firebase

Widely used by industry, A/B testing is a statistical hypothesis testing technique, which answers the question "Is B better than A?" Here A and B could be anything of the same kind: content on a website, design elements on a phone app, or even deep learning models. A/B testing is a really useful feature for us when actively developing a model and discovering how our users respond to different iterations of the model.

Users have been using `my_model_v1` for some time now, and we'd like to see whether the v2 iteration has our users going gaga. We'd like to start slow; maybe just 10% of our users should get v2. For that, we can set up an A/B testing experiment as follows:

1. In Firebase, click the A/B Testing section, and then select "Create experiment" (Figure 13-19).

Figure 13-19. A/B testing screen in Firebase where we can create an experiment

2. Select the Remote Config option.
3. In the Basics section, in the "Experiment name" box, enter the experiment name and an optional description (Figure 13-20), and then click Next.

Figure 13-20. The Basics section of the screen to create a remote configuration experiment

4. In the Targeting section that opens, from the "Target users" drop-down menu, select our app and enter the percentage of target users (Figure 13-21).

Figure 13-21. The Targeting section of the Remote Config screen

5. Select a goal metric that makes sense. We discuss this in a little more detail in the next section.

6. In the variants section (Figure 13-22), create a new parameter called `model_name` that reflects the name of the model a particular user would use The control group gets the default model, which is `my_model_v1`. We also create an additional variant with the name `my_model_v2,` which goes to 10% of the users.

.

Figure 13-22. The Variants section of the Remote Config screen

7. Select Review and then select “Start experiment.” Over time, we can increase the distribution of users using the variant.

Ta-da! Now we have our experiment up and running.

Measuring an experiment

Where do we go from here? We want to give our experiment some time to see how it performs. Depending on the type of experiment, we might want to give it a few days to a few weeks. The success of the experiments can be determined by any number of criteria. Google provides a few metrics out of the box that we can use, as shown in Figure 13-23.

Figure 13-23. Analytics available when setting up an A/B testing experiment

Suppose that we want to maximize our estimated total revenue—after all, we all want to become rich like Jian-Yang. We would measure our revenue from the users who have the experiment turned on against the baseline; that is, the users who do not have the experiment turned on. If our revenue per user increased against the baseline, we would consider the experiment a success. Conversely, we would conclude the opposite if there were no increase/decrease in revenue per user. For successful experiments, we want to slowly roll them out to all users. At that point, it ceases to be an experiment and it "graduates" to become a core offering.

Using the Experiment in Code

Now that we have set up an experiment, let's see how to include it in our application using code. To use the appropriate model within our code, we simply need to access the remote configuration object (`remoteConfig`) and get the model name from it. The model name that we get from the remote configuration object will depend on whether the user is included in the experiment. The following lines of code accomplish that:

```
val remoteConfig = FirebaseRemoteConfig.getInstance()
remoteConfig.fetch()
val modelName = remoteConfig.getString("current_best_model")
val remoteModel = FirebaseCloudModelSource.Builder(modelName)
        .enableModelUpdates(true).build()
FirebaseModelManager.getInstance().registerCloudModelSource(remoteModel)
```

The rest of the code to perform the prediction remains exactly the same as in the previous sections. Our app is now ready to use the correct model as dictated by our experiment.

Using the Experiment in Code

Now that we have set up an experiment, let's see how to include it in our application using code. To use the appropriate model within our code, we simply need to access the remote configuration object (`remoteConfig`) and get the model name from it. The model name that we get from the remote configuration object will depend on whether the user is included in the experiment. The following lines of code accomplish that:

```
val remoteConfig = FirebaseRemoteConfig.getInstance()
remoteConfig.fetch()
val modelName = remoteConfig.getString("current_best_model")
val remoteModel = FirebaseCloudModelSource.Builder(modelName)
        .enableModelUpdates(true).build()
FirebaseModelManager.getInstance().registerCloudModelSource(remoteModel)
```

The rest of the code to perform the prediction remains exactly the same as in the previous sections. Our app is now ready to use the correct model as dictated by our experiment.

TensorFlow Lite on iOS

The previous chapters demonstrate how easy it is to use Apple's Core ML on iOS. We can simply drag and drop a model into Xcode and start making inferences with just a couple of lines of code. In contrast, even looking at basic iOS examples such as those on TensorFlow Lite's repository, it becomes evident that it needs a significant amount of boilerplate code to get even the most basic of apps running. In contrast, TensorFlow Lite used in conjunction with ML Kit is a rather pleasant experience. In addition to being able to use a clean and concise API, we get all the features we detailed earlier in this chapter—remotely downloading and updating models, model A/B testing, and cloud fallback for processing. All of this without having to do much extra work. A developer writing a deep learning app for both iOS and Android might consider using ML Kit as a way to "build once, use everywhere."

TensorFlow Lite on iOS

The previous chapters demonstrate how easy it is to use Apple's Core ML on iOS. We can simply drag and drop a model into Xcode and start making inferences with just a couple of lines of code. In contrast, even looking at basic iOS examples such as those on TensorFlow Lite's repository, it becomes evident that it needs a significant amount of boilerplate code to get even the most basic of apps running. In contrast, TensorFlow Lite used in conjunction with ML Kit is a rather pleasant experience. In addition to being able to use a clean and concise API, we get all the features we detailed earlier in this chapter—remotely downloading and updating models, model A/B testing, and cloud fallback for processing. All of this without having to do much extra work. A developer writing a deep learning app for both iOS and Android might consider using ML Kit as a way to "build once, use everywhere."

Performance Optimizations

In Chapter 6, we explored quantization and pruning, mostly from a theoretical standpoint. Let's see them up close from TensorFlow Lite's perspective and the tools to achieve them.

Performance Optimizations

In Chapter 6, we explored quantization and pruning, mostly from a theoretical standpoint. Let's see them up close from TensorFlow Lite's perspective and the tools to achieve them.

Quantizing with TensorFlow Lite Converter

For iOS, Apple provides `quantization_utils` in its Core ML Tools package. And for TensorFlow Lite, the equivalent is the already built-in `tflite_convert` tool, which we used earlier in this chapter. On the command line, we can specify the input file, the model graph, the data type to convert to, and the names of the input and output (which can be inspected using Netron, as shown in Chapter 11). Going from 32-bit to 8-bit integer representation means a four times smaller model, with relatively little loss in accuracy.

```
$ tflite_convert \
  --output_file=quantized-model.tflite \
  --graph_def_file=/tmp/some-graph.pb \
  --inference_type=QUANTIZED_UINT8 \
  --input_arrays=input \
  --output_arrays=MobilenetV1/Predictions/Reshape_1 \
  --mean_values=128 \
  --std_dev_values=127
```

When it's finished, this command should give us the `quantized-model.tflite` model.

Quantizing with TensorFlow Lite Converter

For iOS, Apple provides `quantization_utils` in its Core ML Tools package. And for TensorFlow Lite, the equivalent is the already built-in `tflite_convert` tool, which we used earlier in this chapter. On the command line, we can specify the input file, the model graph, the data type to convert to, and the names of the input and output (which can be inspected using Netron, as shown in Chapter 11). Going from 32-bit to 8-bit integer representation means a four times smaller model, with relatively little loss in accuracy.

```
$ tflite_convert \
  --output_file=quantized-model.tflite \
  --graph_def_file=/tmp/some-graph.pb \
  --inference_type=QUANTIZED_UINT8 \
  --input_arrays=input \
  --output_arrays=MobilenetV1/Predictions/Reshape_1 \
  --mean_values=128 \
  --std_dev_values=127
```

When it’s finished, this command should give us the `quantized-model.tflite` model.

TensorFlow Model Optimization Toolkit

The TensorFlow Lite converter is the simplest way to get our models quantized. It's worth noting that the converter quantizes the models post-training. Due to the decrease in representation power, there can be a tiny but noticeable loss in accuracy. Could we do better? *Quantization-aware training*, as the name suggests, accounts for the effects of quantization during training time and attempts to compensate and minimize the losses that would have happened in post-training quantization.

Although both forms of quantization offer 75% reduction in the size of the model, experiments have shown the following:

- In the case of MobileNetV2, compared to an eight-point loss in accuracy with post-training quantization, quantization-aware training yielded only a one-point loss.
- For InceptionV3, quantization-aware training yielded a whopping 52% reduction in latency, compared to 25% reduction with post-training quantization.

> **NOTE**
>
> It is worth noting that these accuracy metrics are on the 1,000 class ImageNet test set. Most problems have less complexity with a smaller number of classes. Post-training quantization should result in a smaller loss on such simpler problems.

Quantization-aware training can be implemented with the TensorFlow Model Optimization Toolkit. This toolkit additional offers an increasing array of tools for model compression, including pruning. Additionally, the TensorFlow Lite model repository already offers these prequantized models using this technique. Table 13-2 lists the effects of various quantization strategies.

Table 13-2. Effects of different quantization strategies (8-bit) on models (source: TensorFlow Lite Model optimization documentation)

Model	MobileNet	MobileNetV2	InceptionV3

Top-1 accuracy	**Original**	0.709	0.719	0.78
	Post-training quantized	0.657	0.637	0.772
	Quantization-aware training	0.7	0.709	0.775
Latency (ms)	**Original**	124	89	1130
	Post-training quantized	112	98	845
	Quantization-aware training	64	54	543
Size (MB)	**Original**	16.9	14	95.7
	Optimized	4.3	3.6	23.9

Top-1 accuracy	**Original**	0.709	0.719	0.78
	Post-training quantized	0.657	0.637	0.772
	Quantization-aware training	0.7	0.709	0.775
Latency (ms)	**Original**	124	89	1130
	Post-training quantized	112	98	845
	Quantization-aware training	64	54	543
Size (MB)	**Original**	16.9	14	95.7
	Optimized	4.3	3.6	23.9

Fritz

As we have seen so far, the primary purpose of Core ML and TensorFlow Lite is to serve fast mobile inferences. Have a model, plug it into the app, and run inferences. Then came ML Kit, which apart from its built-in AI capabilities, made it easier to deploy models and monitor our custom models (with Firebase). Taking a step back to look at the end-to-end pipeline, we had training, conversion to mobile format, optimizing its speed, deploying to users, monitoring the performance, keeping track of model versions. These are several steps spread across many tools. And there is a good chance that there is no single owner of the entire pipeline. This is because oftentimes these tools require some level of familiarity (e.g., having to deploy a model to users might be outside a data scientist's comfort zone). Fritz, a Boston-based startup, is attempting to bring down these barriers and make the full cycle from model development to deployment even more straightforward for both data scientists and mobile developers.

Fritz offers an end-to-end solution for mobile AI development that includes the following noteworthy features:

- Ability to deploy models directly to user devices from Keras after training completes using callbacks.
- Ability to benchmark a model directly from the computer without having to deploy it on a phone. The following code demonstrates this:

```
$ fritz model benchmark <path to keras model.h5>
 ...
 ------------------------
 Fritz Model Grade Report
 ------------------------

 Core ML Compatible:              True
 Predicted Runtime (iPhone X):    31.4 ms (31.9 fps)
 Total MFLOPS:                    686.90
 Total Parameters:                1,258,580
```

```
Fritz Version ID:                <Version UID>
```

- Ability to encrypt models so that our intellectual property can be protected from theft from a device by nefarious actors.
- Ability to implement many advanced computer-vision algorithms with just a few lines of code. The SDK comes with ready-to-use, mobile-friendly implementations of these algorithms, which run at high frame rates. For example, image segmentation, style transfer, object detection, and pose estimation. Figure 13-24 shows benchmarks of object detection run on various iOS devices.

```ios
let poseModel = FritzVisionPoseModel()
guard let poseResult = try? poseModel.predict(image) else { return }
let imageWithPose = poseResult.drawPose() // Overlays pose on input.
```

Figure 13-24. Performance of Fritz SDK's object detection functionality on different mobile devices relative to the iPhone X

- Ability to customize the prebuilt models to custom datasets by retraining using their Jupyter notebooks. It's worth noting that this can be a difficult problem (even for professional data scientists) that is simplified significantly because the developer has only to ensure that the data is in the

right format.

- Ability to manage all model versions from the command line.
- Ability to test out mobile readiness of our models using Fritz' open source app called Heartbeat (also available on iOS/Android app stores). Assuming that we have a model ready without much mobile know how, we can clone the app, swap the existing model with our own, and get to see it run on the phone.
- A vibrant community of contributors blogging about the latest in mobile AI on *heartbeat.fritz.ai*.

FROM THE CREATOR'S DESK

By Dan Abdinoor, CEO and cofounder of Fritz

Our motivation to build Fritz came from two somewhat contradictory feelings—inspiration and frustration. The inspiration came from seeing how many previously impossible problems were being solved with machine learning. The combination of data, compute, and model architectures were all starting to click. At the same time, we found ourselves frustrated by the lack of resources for deploying and maintaining machine learning anywhere other than the cloud. For reasons of performance, privacy, and cost, it makes more sense to shift inference to devices where data is being collected, as opposed to moving all that data to a cloud environment, processing it, and then sending predictions back down to the devices. This led us to build Fritz as a platform for developers to collect, train, deploy, and manage machine learning models for edge devices—like mobile phones.

In many ways, the process of building a solution with machine learning is like a race with hurdles. We want to lower or remove those hurdles so that engineers, even those without deep learning or mobile expertise, can use what we built to make great mobile applications. One can get started quickly with efficient pretrained models, or custom train a new one. Our tools should make it easier to deploy models, understand how they are performing, and keep them updated and protected. The results of all this, in part, are models that perform well on edge devices.

Making it easier to implement machine learning features in mobile apps will lead to more powerful, personalized experiences that retain current users, attract new ones, and shift our thinking about what’s possible. Ultimately, we believe machine learning is the most fundamental change to computer science in decades, and we want to help build, educate, and evolve the community.

Making it easier to implement machine learning features in mobile apps will lead to more powerful, personalized experiences that retain current users, attract new ones, and shift our thinking about what’s possible. Ultimately, we believe machine learning is the most fundamental change to computer science in decades, and we want to help build, educate, and evolve the community.

A Holistic Look at the Mobile AI App Development Cycle

Until this point in the book, we've looked at a bunch of techniques and technologies that enable us to perform individual tasks in the mobile AI development cycle. Here, we tie everything together by exploring the kind of questions that come up throughout this life cycle. Figure 13-25 provides a broad overview of the development cycle.

Figure 13-25. Mobile AI app development life cycle

A Holistic Look at the Mobile AI App Development Cycle

Until this point in the book, we've looked at a bunch of techniques and technologies that enable us to perform individual tasks in the mobile AI development cycle. Here, we tie everything together by exploring the kind of questions that come up throughout this life cycle. Figure 13-25 provides a broad overview of the development cycle.

Figure 13-25. Mobile AI app development life cycle

How Do I Collect Initial Data?

We can apply a few different strategies to accomplish this:

- Find the object of interest and manually take photos with different angles, lighting, environments, framing, and so on.
- Scrape from the internet using browser extensions like Fatkun (Chapter 12)
- Find existing datasets (Google Dataset Search). For example, Food-101 for dishes.
- Synthesize your own dataset.

    1. Place the object (foreground) in front of a green screen (background) and take a photograph of it. Replace the background with random images to synthesize a large dataset, while zooming, cropping, and rotating to create potentially hundreds of images.
    2. If a green screen is not possible, segment (i.e., cut) the object out of the background from existing real-world images and repeat the previous step in order to build a robust, diverse dataset. It's important to note that, in points (a) and (b), there needs to be sufficient diversity in the foreground; otherwise, the network might overlearn that one example instead of understanding the object.
    3. Find a realistic 3D model of the object of interest and place it in realistic environments using a 3D framework such as Unity. Adjust the lighting and camera position, zoom, and rotation to take snapshots of this object from many angles. We explored companies such as AI.Reverie and CVEDIA in Chapter 7 that work in this space. We use photo-realistic simulators to train models for the self-driving chapters (Chapter 16 and Chapter 17).

How Do I Collect Initial Data?

We can apply a few different strategies to accomplish this:

- Find the object of interest and manually take photos with different angles, lighting, environments, framing, and so on.
- Scrape from the internet using browser extensions like Fatkun (Chapter 12).
- Find existing datasets (Google Dataset Search). For example, Food-101 for dishes.
- Synthesize your own dataset.

    1. Place the object (foreground) in front of a green screen (background) and take a photograph of it. Replace the background with random images to synthesize a large dataset, while zooming, cropping, and rotating to create potentially hundreds of images.
    2. If a green screen is not possible, segment (i.e., cut) the object out of the background from existing real-world images and repeat the previous step in order to build a robust, diverse dataset. It's important to note that, in points (a) and (b), there needs to be sufficient diversity in the foreground; otherwise, the network might overlearn that one example instead of understanding the object.
    3. Find a realistic 3D model of the object of interest and place it in realistic environments using a 3D framework such as Unity. Adjust the lighting and camera position, zoom, and rotation to take snapshots of this object from many angles. We explored companies such as AI.Reverie and CVEDIA in Chapter 7 that work in this space. We use photo-realistic simulators to train models for the self-driving chapters (Chapter 16 and Chapter 17).

How Do I Label My Data?

Most of the steps in the previous answer should have already given you labels for the data. For unlabeled collections, use labeling tools such as Supervisely, Labelbox, and Diffgram. For really large datasets in which annotating data by yourself is not feasible, socially responsible labeling services such as Digital Data Divide, iMerit, and Samasource, which provide income opportunities to disadvantaged populations, might be a good option.

How Do I Label My Data?

Most of the steps in the previous answer should have already given you labels for the data. For unlabeled collections, use labeling tools such as Supervisely, Labelbox, and Diffgram. For really large datasets in which annotating data by yourself is not feasible, socially responsible labeling services such as Digital Data Divide, iMerit, and Samasource, which provide income opportunities to disadvantaged populations, might be a good option.

How Do I Train My Model?

The following are the two broad approaches to train a model:

- With code: Use Keras and TensorFlow (Chapter 3).
- Without code: Use custom classifier services such as Google's Auto ML, Microsoft's CustomVision.ai, and Clarifai (benchmarked in Chapter 8), or Apple's ecosystem-only Create ML (Chapter 12).

How Do I Train My Model?

The following are the two broad approaches to train a model:

- With code: Use Keras and TensorFlow (Chapter 3).
- Without code: Use custom classifier services such as Google's Auto ML, Microsoft's CustomVision.ai, and Clarifai (benchmarked in Chapter 8), or Apple's ecosystem-only Create ML (Chapter 12).

How Do I Convert the Model to a Mobile-Friendly Format?

The following are a few different ways to convert a model to a mobile-compatible format:

- Use Core ML Tools (Apple only).
- Use TensorFlow Lite Converter for iOS and Android.
- Alternatively, use Fritz for the end-to-end pipeline.

How Do I Convert the Model to a Mobile-Friendly Format?

The following are a few different ways to convert a model to a mobile-compatible format:

- Use Core ML Tools (Apple only).
- Use TensorFlow Lite Converter for iOS and Android.
- Alternatively, use Fritz for the end-to-end pipeline.

How Do I Make my Model Performant?

Here are some techniques to make a model performant:

- Start with an efficient model such as the MobileNet family, or even better, EfficientNet.
- Make the model smaller in size by quantizing and pruning it to improve loading and inference times, while keeping the model accuracy relatively intact (Chapter 6, Chapter 11, and Chapter 13). Expect up to a 75% reduction in size with little loss in accuracy.
- Use Core ML Tools for the Apple ecosystem or TensorFlow Model Optimization Kit for both iOS and Android.

How Do I Make my Model Performant?

Here are some techniques to make a model performant:

- Start with an efficient model such as the MobileNet family, or even better, EfficientNet.
- Make the model smaller in size by quantizing and pruning it to improve loading and inference times, while keeping the model accuracy relatively intact (Chapter 6, Chapter 11, and Chapter 13). Expect up to a 75% reduction in size with little loss in accuracy.
- Use Core ML Tools for the Apple ecosystem or TensorFlow Model Optimization Kit for both iOS and Android.

How Do I Build a Great UX for My Users?

Obviously, that's going to depend on the kind of problem you are trying to solve But a general guideline is to strike the proper balance between interactivity, performance, and resource usage (memory, CPU, battery, etc.). And, of course, an intelligent feedback mechanism that enables effortless data collection and feedback helps. Gamifying this experience would take it to an entirely new level

In the case of the food classifier app, after the user takes a picture, our UI would show a list of top five candidate predictions (in decreasing order of confidence) for a given photo. For example, if the user takes a picture of a pizza, the candidate predictions shown on screen could be "pie - 75%," "pizza - 15%," "naan - 6%," "bread - 3%," "casserole - 1%." In this case, the user would have selected the second prediction. For a perfect model, the user would always select the first prediction. Because our model is not perfect, the rank of the prediction the user selects becomes a signal that will help improve the model in the future. In the absolute worst case, in which none of the predictions were correct, the user should have a way to manually label the data. Providing an autosuggest feature during manual entry will help keep clean labels in the dataset.

An even better experience could be one in which the user never needs to click a picture. Rather, the predictions are available in real time.

How Do I Build a Great UX for My Users?

Obviously, that's going to depend on the kind of problem you are trying to solve. But a general guideline is to strike the proper balance between interactivity, performance, and resource usage (memory, CPU, battery, etc.). And, of course, an intelligent feedback mechanism that enables effortless data collection and feedback helps. Gamifying this experience would take it to an entirely new level.

In the case of the food classifier app, after the user takes a picture, our UI would show a list of top five candidate predictions (in decreasing order of confidence) for a given photo. For example, if the user takes a picture of a pizza, the candidate predictions shown on screen could be "pie - 75%," "pizza - 15%," "naan - 6%," "bread - 3%," "casserole - 1%." In this case, the user would have selected the second prediction. For a perfect model, the user would always select the first prediction. Because our model is not perfect, the rank of the prediction the user selects becomes a signal that will help improve the model in the future. In the absolute worst case, in which none of the predictions were correct, the user should have a way to manually label the data. Providing an autosuggest feature during manual entry will help keep clean labels in the dataset.

An even better experience could be one in which the user never needs to click a picture. Rather, the predictions are available in real time.

How Do I Make the Model Available to My Users?

The following are some ways to deploy models to users:

- Bundle the model into the app binary and release it on the app stores.
- Alternatively, host the model in the cloud and have the app download the model, as necessary.
- Use a model management service like Fritz or Firebase (integrated with MI Kit).

How Do I Make the Model Available to My Users?

The following are some ways to deploy models to users:

- Bundle the model into the app binary and release it on the app stores.
- Alternatively, host the model in the cloud and have the app download the model, as necessary.
- Use a model management service like Fritz or Firebase (integrated with ML Kit).

How Do I Measure the Success of My Model?

The very first step is to determine the success criteria. Consider the following examples:

- “My model should run inferences in under 200 ms at the 90th percentile.”
- “Users who use this model open the app every day.”
- In the food classifier app, a success metric could be something along the lines of “80% of users selected the first prediction in the list of predictions.’

These success criteria are not set in stone and should continue to evolve over time. It’s very essential to be data driven. When you have a new model version, run A/B tests on a subset of users and evaluate the success criteria against that version to determine whether it’s an improvement over the previous version.

How Do I Measure the Success of My Model?

The very first step is to determine the success criteria. Consider the following examples:

- “My model should run inferences in under 200 ms at the 90th percentile.”
- “Users who use this model open the app every day.”
- In the food classifier app, a success metric could be something along the lines of “80% of users selected the first prediction in the list of predictions.”

These success criteria are not set in stone and should continue to evolve over time. It’s very essential to be data driven. When you have a new model version, run A/B tests on a subset of users and evaluate the success criteria against that version to determine whether it’s an improvement over the previous version.

How Do I Improve My Model?

Here are some ways to improve the quality of our models:

- Collect feedback on individual predictions from users: what was correct, and, more important, what was incorrect. Feed these images along with the corresponding labels as input for the next model training cycle. Figure 13-26 illustrates this.
- For users who have explicitly opted in, collect frames automatically whenever the prediction confidence is low. Manually label those frames and feed them into the next training cycle.

Figure 13-26. The feedback cycle of an incorrect prediction, generating more training data, leading to an improved model

1

How Do I Update the Model on My Users' Phones?

Here are some ways to update the model on your users' phones:

- Bundle the new model into the next app release. This is slow and inflexible
- Host the new model on the cloud and force the apps out in the world to download the new model. Prefer to do this when the user is on WiFi.
- Use a model management system such as Firebase (in conjunction with ML Kit) or Fritz to automate a lot of the grunt work involved.

With all of these questions answered, let's appreciate the beauty of how this app improves on its own.

How Do I Update the Model on My Users' Phones?

Here are some ways to update the model on your users' phones:

- Bundle the new model into the next app release. This is slow and inflexible.
- Host the new model on the cloud and force the apps out in the world to download the new model. Prefer to do this when the user is on WiFi.
- Use a model management system such as Firebase (in conjunction with ML Kit) or Fritz to automate a lot of the grunt work involved.

With all of these questions answered, let's appreciate the beauty of how this app improves on its own.

The Self-Evolving Model

For applications that need only mature pretrained models, our job is already done. Just plug the model into an app and we're ready to go. For custom-trained models that rely especially on scarce training data, we can get the user involved in building a self-improving, ever-evolving model.

At the most fundamental level, each time a user uses the app, they're providing the necessary feedback (image + label) to improve the model further, as illustrated in Figure 13-27.

Figure 13-27. The self-evolving model cycle

Just like a university student must graduate multiple years before getting ready to step into the real world, a model needs to go through phases of development before it reaches its final users. The following is a popular model of phasing releases in software, which is also a useful guideline for releasing AI models.

1. Dev

This is the initial phase of development in which the developers of the app

are the only users of it. Data accumulation is slow, the experience is very buggy, and model predictions can be quite unreliable.

2. Alpha

After the app is ready to be tested by a few users beyond the developers, it is now considered to be in alpha. User feedback is extremely crucial here because it would serve to improve the app experience as well provide data at a greater scale to the pipeline. The experience is not nearly as buggy, and the model predictions are a little more reliable here. In some organizations, this phase is also known as *dogfooding* (i.e., internal application testing by employees).

3. Beta

An app in beta has significantly more users than in alpha. User feedback is also quite important here. The data collection rate is at an even larger scale. Many different devices are out in the real world collecting data to help improve the model rapidly. The experience is a lot more stable, and the model predictions are a lot more reliable, thanks to the alpha users. Beta apps tend to be hosted on Apple's TestFlight for iOS and Google Play Console for Android. Third-party services such as HockeyApp and TestFairy are also popular for hosting beta programs.

4. Prod

An app in production is stable and is widely available. Although the data is coming in at large rates, the model is fairly stable at this point, and it does not learn as much. However, as real-world usage keeps growing, it's able to get better at edge cases it might not have been seen before in the first three phases. When the model is mature enough, small version improvements can be alpha/beta tested or A/B tested on subsets of the production audience.

Although many data scientists assume the data to be available before starting development, people in the mobile AI world might need to bootstrap on a small dataset and improve their system incrementally.

With this self-evolving system for the Shazam for Food app, the most difficult choice that the developers must make should be the restaurants they visit to collect their seed data.

S

Case Studies

Let’s look at a few interesting examples that show how what we have learned so far is applied in the industry.

Case Studies

Let’s look at a few interesting examples that show how what we have learned so far is applied in the industry.

Lose It!

We need to confess here. The app we were building all along in this chapter already exists. Erlich Bachman's dream was already built a year before by a Boston-based company named Fit Now. Lose It! (Figure 13-28) claims to have helped 30 million users lose 85 million pounds by tracking what they eat. Users can point their phone's camera at barcodes, nutritional labels, and the food that they're just about to eat to track calories and macros they're consuming for every single meal.

The company first implemented its food scanning system with a cloud-based algorithm. Due to the resource demands and network delay, the experience could not be real time. To improve the experience, the Fit Now team migrated its models to TensorFlow Lite to optimize them for mobile and used ML Kit to seamlessly deploy to its millions of users. With a constant feedback loop, model updates, and A/B testing, the app keeps improving essentially on its own.

1

Cancel
Done
920cals
Snap It BETA suggestions from photo
Pizza
Garlic Bread
Pie
Omelette
Lasagna
Pizza, Roasted Veggies and Chi...
4 Each
920
Add foods in this photo

Figure 13-28. Snap It feature from Lose It! showing multiple suggestions for a scanned food item

Figure 13-28. Snap It feature from Lose It! showing multiple suggestions for a scanned food item

Portrait Mode on Pixel 3 Phones

One of the key visual concepts that separates professional photographers from the amateur wannabes is bokeh, or blurring of the background behind the subject of the image. (Not to be confused with the Python visualization tool of the same name.) The closer the camera is located to the subject, the more the background appears blurred. With the help of professional lenses with a low *f*-stop number (which often run into the thousands of dollars), one can produce spectacular blurs.

But if you don't have that kind of money, AI can help. Google's Pixel 3 offers a "Portrait" mode that creates the bokeh effect. Essentially, it uses a CNN to estimate the depth of each pixel in a scene, and determine which pixels belong to the foreground and the background. The background pixels are then blurred to varying intensities to create the bokeh effect, as demonstrated in Figure 13-29.

Depth estimation is a compute-intensive task so running it fast is essential. Tensorflow Lite with a GPU backend comes to the rescue, with a roughly 3.5 times speedup over a CPU backend.

Google accomplished this by training a neural network that specializes in depth estimation. It used a rig with multiple phone cameras (which it termed "Frankenphone") to take pictures of the same scene from slightly different viewpoints belonging to each camera. It then used the *parallax effect* experienced by each pixel to precisely determine the depth of each pixel. This depth map was then fed into the CNN along with the images.

t

)

Figure 13-29. Portrait effect on Pixel 3, which achieves separation between foreground and background using blurring

Figure 13-29. Portrait effect on Pixel 3, which achieves separation between foreground and background using blurring

Speaker Recognition by Alibaba

Imagine speaking “Hey Siri” or “Hey Google” into a friend’s phone, followed by the magical words “read the last text.” From a privacy standpoint, it would be terrifying if this worked. Obviously, most mobile operating systems today ask us to unlock the phone first before proceeding. This creates a not-so-smooth experience for the owner of the phone.

Alibaba Machine Intelligence Labs solved this problem using the following approach. First, by converting speech into images of spectrograms (extracting audio features with mel-frequency cepstrum algorithm), training a CNN (using transfer learning), and finally deploying it to devices using Tensorflow Lite. That’s right: CNNs can do more than just computer vision!

By recognizing speakers, they are able to personalize the content on devices with many users in a family (e.g., in its “Talk to your TV” feature on its Netflix-like TV app). Additionally, by recognizing whether the sound is a human voice, it is able to isolate the spoken commands to transcribe from background noise. To make processing faster with Tensorflow Lite, engineers kept the `USE_NEON` flag to accelerate instruction sets for ARM-based CPUs. The team reported a speed increase of four times with this optimization.

Speaker Recognition by Alibaba

Imagine speaking “Hey Siri” or “Hey Google” into a friend’s phone, followed by the magical words “read the last text.” From a privacy standpoint, it would be terrifying if this worked. Obviously, most mobile operating systems today ask us to unlock the phone first before proceeding. This creates a not-so-smooth experience for the owner of the phone.

Alibaba Machine Intelligence Labs solved this problem using the following approach. First, by converting speech into images of spectrograms (extracting audio features with mel-frequency cepstrum algorithm), training a CNN (using transfer learning), and finally deploying it to devices using Tensorflow Lite. That’s right: CNNs can do more than just computer vision!

By recognizing speakers, they are able to personalize the content on devices with many users in a family (e.g., in its “Talk to your TV” feature on its Netflix-like TV app). Additionally, by recognizing whether the sound is a human voice, it is able to isolate the spoken commands to transcribe from background noise. To make processing faster with Tensorflow Lite, engineers kept the `USE_NEON` flag to accelerate instruction sets for ARM-based CPUs. The team reported a speed increase of four times with this optimization.

Face Contours in ML Kit

Want to quickly build a Snapchat face filter–like functionality without getting a Ph.D.? ML Kit also provides the goods to do that—an API to identify facial contours, a set of points (in x and y coordinates) that follow the shape of facial features in the input image. In total, 133 points map to the various facial contours, including 16 points representing each eye, while 36 points map to the oval shape around the face, as demonstrated in Figure 13-30.

Internally, ML Kit is running a deep learning model using TensorFlow Lite. In experiments with the new TensorFlow Lite GPU backend, Google found an increase in speed of four times on Pixel 3 and Samsung Galaxy S9, and a six-times speedup on iPhone 7, compared to the previous CPU backend. The effect of this is that we can precisely place a hat or sunglasses on a face in real time.

Figure 13-30. 133 Face contour points identified by ML Kit (image source)

Figure 13-30. 133 Face contour points identified by ML Kit (image source)

Real-Time Video Segmentation in YouTube Stories

Green screens are staple equipment for anyone in the video production industry. By choosing a color not matching the subject, the background can be changed during postproduction using the *chroma keying* technique. Obviously, this requires expensive software and powerful machines for postproduction. Many YouTubers have the same needs but might not have the budget. And now they have a solution within the YouTube Stories app—a real-time video segmentation option implemented over TensorFlow Lite.

The key requirements here are first, to run semantic segmentation fast (more than 30 frames per second), and second, be temporally consistent; for instance, achieving smooth frame-to-frame temporal continuity on the edges. If we attempt to run semantic segmentation over multiple frames, we'll quickly notice the edges of many of the segmented masks bouncing around. The key trick employed here is to pass the segmented masks of the person's face to the next frame as a prior. Because we traditionally work on three channels (RGB), the trick is to add a fourth channel, which essentially is the output of the previous frame, as illustrated in Figure 13-31.

Figure 13-31. An input image (left) is broken down into its three components layers (R, G, B). The output mask of the previous frame is then appended with these components (image source)

With other optimizations to the structure of the base CNN, the YouTube team was able to run the system at more than 100 frames per second on an iPhone 7, and faster than 40 frames per second on a Pixel 2. Additionally, comparing the CPU versus GPU backend of TensorFlow Lite, an increase in speed of 18 times is observed when choosing the GPU backend over the CPU backend (much more acceleration compared to the usual two to seven times for other semantic segmentation tasks for images only).

ı

e

Summary

In this chapter, we discussed the architecture of TensorFlow Lite, as it relates to Android's Neural Network API. We then worked on getting a simple object recognition app running on an Android device. We covered Google's ML Kit and talked about the reasons why you might want to use it. We additionally went through how to convert our TensorFlow models to TensorFlow Lite format so that they could be used within an Android device. We finally discussed a couple of examples of how TensorFlow Lite is being used to solve real-world problems.

In the next chapter, we look at how we can use real-time deep learning to develop an interactive, real-world application.

Summary

In this chapter, we discussed the architecture of TensorFlow Lite, as it relates to Android's Neural Network API. We then worked on getting a simple object recognition app running on an Android device. We covered Google's ML Kit and talked about the reasons why you might want to use it. We additionally went through how to convert our TensorFlow models to TensorFlow Lite format so that they could be used within an Android device. We finally discussed a couple of examples of how TensorFlow Lite is being used to solve real-world problems.

In the next chapter, we look at how we can use real-time deep learning to develop an interactive, real-world application.

Chapter 14. Building the Purrfect Cat Locator App with TensorFlow Object Detection API

Bob gets frequent visits from a neighborhood feral cat. Those visits result in less than pleasant outcomes. You see, Bob has a nice fairly large garden that he tends to with great care. However, this furry little fella visits his garden every night and starts chewing on a bunch of plants. Months of hard work gets destroyed overnight. Clearly unhappy with this situation, Bob is keen on doing something about it.

Channeling his inner Ace Ventura (Pet Detective), he tries to stay awake at night to drive the cat away, but clearly, that's not sustainable in the long term. After all, Red Bull has its limits. Reaching a breaking point, he decides to use the nuclear option: use AI in conjunction with his sprinkler system to literally "turn the hose" on the cat.

In his sprawling garden, he sets up a camera that can track the motion of the cat and turn on the nearest set of sprinklers to scare the cat away. He has an old phone lying around, and all he needs is a way to determine the cat's location in real time, so he can control accordingly which sprinkler to trigger.

Hopefully, after a few unwelcome baths, as demonstrated in Figure 14-1, the cat would be disincentivized from raiding Bob's garden.

Figure 14-1. Building an AI cat sprinkler system, like this Havahart Spray Away Motion Detector

TIP

Keen observers might have noticed that we're attempting to find the position of the cat in 2D space, whereas the cat itself exists in 3D space. We can make assumptions to simplify the problem of locating the cat in real-world coordinates by assuming that the camera will always be in a fixed position. To determine the distance of the cat, we can use the size of an average cat to take measurements of its apparent size on a picture at regular intervals of distance from the camera. For example, an average cat might appear to have a bounding box height of 300 pixels at a distance of two feet from the camera, whereas it might occupy 250 pixels when it's three feet away from the camera lens.

In this chapter, we help Bob make his cat detector. (Note to animal-rights advocates: no cats were harmed in the making of this chapter.) We answer the following questions along the way:

- What are the different kinds of computer-vision tasks?
- How do I reuse a model pretrained for an existing class of objects?
- Can I train an object detector model without writing any code?
- I want more granular control for greater accuracy and speed. How do I train a custom object detector?

Figure 14-1. Building an AI cat sprinkler system, like this Havahart Spray Away Motion Detector

TIP

Keen observers might have noticed that we're attempting to find the position of the cat in 2D space, whereas the cat itself exists in 3D space. We can make assumptions to simplify the problem of locating the cat in real-world coordinates by assuming that the camera will always be in a fixed position. To determine the distance of the cat, we can use the size of an average cat to take measurements of its apparent size on a picture at regular intervals of distance from the camera. For example, an average cat might appear to have a bounding box height of 300 pixels at a distance of two feet from the camera, whereas it might occupy 250 pixels when it's three feet away from the camera lens.

In this chapter, we help Bob make his cat detector. (Note to animal-rights advocates: no cats were harmed in the making of this chapter.) We answer the following questions along the way:

- What are the different kinds of computer-vision tasks?
- How do I reuse a model pretrained for an existing class of objects?
- Can I train an object detector model without writing any code?
- I want more granular control for greater accuracy and speed. How do I train a custom object detector?

Types of Computer-Vision Tasks

In most of the previous chapters, we've essentially looked at one kind of problem: object classification. In that, we found out whether an image contained objects of a certain class. In Bob's scenario, in addition to knowing whether the camera is seeing a cat, it is necessary to know exactly where the cat is in order to trigger the nearest sprinkler. Determining the location of an object within an image is a different type of problem: *object detection*. Before we jump into object detection, let's look at the variety of frequent computer-vision tasks and the questions they purport to answer.

Types of Computer-Vision Tasks

In most of the previous chapters, we've essentially looked at one kind of problem: object classification. In that, we found out whether an image contained objects of a certain class. In Bob's scenario, in addition to knowing whether the camera is seeing a cat, it is necessary to know exactly where the cat is in order to trigger the nearest sprinkler. Determining the location of an object within an image is a different type of problem: *object detection*. Before we jump into object detection, let's look at the variety of frequent computer-vision tasks and the questions they purport to answer.

Classification

We have looked at several examples of classification throughout this book. Simply put, the task of classification assigns an image with the class(es) of the object(s) present in the image. It answers the question: “Is there an object of class X in the image?” Classification is one of the most fundamental computer-vision tasks. In an image, there might be objects of more than one class—called multiclass classification. The label in this case for one image would be a list of all object classes that the image contains.

Classification

We have looked at several examples of classification throughout this book. Simply put, the task of classification assigns an image with the class(es) of the object(s) present in the image. It answers the question: “Is there an object of class X in the image?” Classification is one of the most fundamental computer-vision tasks. In an image, there might be objects of more than one class—called multiclass classification. The label in this case for one image would be a list of all object classes that the image contains.

Localization

The downside of classification is that it does not tell us where in the image a particular object is, how big or small it is, or how many objects there are. It only tells us that an object of a particular class exists somewhere within it. *Localization*, otherwise known as *classification with localization*, can inform us which classes are in an image and where they are located within the image. The keyword here is *classes*, as opposed to individual objects, because localization gives us only one bounding box per class. Just like classification, it cannot answer the question “How many objects of class X are in this image?”

As you will see momentarily, localization will work correctly only when it is guaranteed that there is a single instance of each class. If there is more than one instance of a class, one bounding box might encompass some or all objects of that class (depending upon the probabilities). But just one bounding box per class seems rather restrictive, doesn’t it?

Localization

The downside of classification is that it does not tell us where in the image a particular object is, how big or small it is, or how many objects there are. It only tells us that an object of a particular class exists somewhere within it. *Localization*, otherwise known as *classification with localization*, can inform us which classes are in an image and where they are located within the image. The keyword here is *classes*, as opposed to individual objects, because localization gives us only one bounding box per class. Just like classification, it cannot answer the question “How many objects of class X are in this image?”

As you will see momentarily, localization will work correctly only when it is guaranteed that there is a single instance of each class. If there is more than one instance of a class, one bounding box might encompass some or all objects of that class (depending upon the probabilities). But just one bounding box per class seems rather restrictive, doesn’t it?

Detection

When we have multiple objects belonging to multiple classes in the same image, localization will not suffice. Object detection would give us a bounding rectangle for every instance of a class, for all classes. It also informs us as to the class of the object within each of the detected bounding boxes. For example, in the case of a self-driving car, object detection would return one bounding box per car, person, lamp post, and so on that the car sees. Because of this ability, we also can use it in applications that need to count. For example, counting the number of people in a crowd.

Unlike localization, detection does not have a restriction on the number of instances of a class in an image. This is why it is used in real-world applications.

WARNING

Often people use the term “object localization” when, really, they mean “object detection.” It’s important to note the distinction between the two.

Detection

When we have multiple objects belonging to multiple classes in the same image, localization will not suffice. Object detection would give us a bounding rectangle for every instance of a class, for all classes. It also informs us as to the class of the object within each of the detected bounding boxes. For example, in the case of a self-driving car, object detection would return one bounding box per car, person, lamp post, and so on that the car sees. Because of this ability, we also can use it in applications that need to count. For example, counting the number of people in a crowd.

Unlike localization, detection does not have a restriction on the number of instances of a class in an image. This is why it is used in real-world applications.

WARNING

Often people use the term “object localization” when, really, they mean “object detection.” It’s important to note the distinction between the two.

Segmentation

Object segmentation is the task of assigning a class label to individual pixels throughout an image. A children's coloring book where we use crayons to color various objects is a real-world analogy. Considering each pixel in the image is being assigned a class, it is a highly compute-intensive task. Although object localization and detection give bounding boxes, segmentation produces groups of pixels—also known as *masks*. Segmentation is obviously much more precise about the boundaries of an object compared to detection. For example, a cosmetic app that changes a user's hair color in real-time would use segmentation. Based on the types of masks, segmentation comes in different flavors.

Semantic segmentation

Given an image, *semantic segmentation* assigns one mask per class. If there are multiple objects of the same class, all of them are assigned to the same mask. There is no differentiation between the instances of the same class. Just like localization, semantic segmentation cannot count.

Instance-level segmentation

Given an image, *instance-level segmentation* identifies the area occupied by each instance of each class. Different instances of the same class will be segmented uniquely.

Table 14-1 lists the different computer-vision tasks.

Table 14-1. Types of computer-vision tasks illustrated using image ID 120524 from the MS COCO dataset

Task	Image	In lay terms	Outputs
Object classification		Is there a sheep in this image?	Class probabilities

Object localization		Is there “a” sheep in the image and where is it?	Bounding box and class probability
Object detection		What and where are “all” the objects in this image?	Bounding boxes, class probabilities, class ID prediction
Semantic segmentation		Which pixels in this image belong to different classes; e.g., the class “sheep,” “dog,” “person”?	One mask per class
Instance-level segmentation		Which pixels in this image belong to each instance of each class; e.g., “sheep,” “dog,” “person”?	One mask per instance of each class

Approaches to Object Detection

Depending on the scenario, your needs, and technical know-how, there are a few ways to go about getting the object detection functionality in your applications. Table 14-2 takes a look at some approaches.

Table 14-2. Different approaches to object detection and their trade-offs

	Custom classes	Effort	Cloud or local	Pros and cons
Cloud-based object detection APIs	No	<5 minutes	Cloud only	+ Thousands of classes + State of the art + Fast setup + Scalable API – No customization – Latency due to network
Pretrained model	No	<15 minutes	Local	+ ~100 classes + Choice of model to fit speed and accuracy needs + Can run on the edge – No customization
Cloud-based model training	Yes	<20 minutes	Cloud and local	+ GUI-based, no coding for training + Customizability + Choice between powerful model (for cloud) and efficient model (for edge) + Scalable API – Uses transfer learning, might not allow full-model retraining
Custom training a model	Yes	4 hours to 2 days	Local	+ Highly customizable + Largest variety of models available for different speed and accuracy requirements – Time consuming and highly involved

It’s worth noting that the options marked “Cloud” are already engineered for scalability. At the same time, the options marked as being available for “Local” inferences can also be deployed on the cloud. To achieve high scale, we can host them on the cloud similar to how we did so for the classification models in Chapter 9.

It’s worth noting that the options marked “Cloud” are already engineered for scalability. At the same time, the options marked as being available for “Local” inferences can also be deployed on the cloud. To achieve high scale, we can host them on the cloud similar to how we did so for the classification models in Chapter 9.

Invoking Prebuilt Cloud-Based Object Detection APIs

As we already saw in Chapter 8, calling cloud-based APIs is relatively straightforward. The process involves setting up an account, getting an API key, reading through the documentation, and writing a REST API client. To make this process easier, many of these services provide Python-based (and other language) SDKs. The main cloud-based object detection API providers are Google's Vision AI (Figure 14-2) and Microsoft's Cognitive Services.

The main advantages of using cloud-based APIs are that they are scalable out of the box and they support recognition of thousands of classes of objects. These cloud giants built their models using large proprietary datasets that are constantly growing, resulting in a very rich taxonomy. Considering that the number of classes in the largest public datasets with object detection labels is usually only a few hundred, there exists no nonproprietary solution out there that is capable of detecting thousands of classes.

The main limitation of cloud-based APIs, of course, is the latency due to network requests. It's not possible to power real-time experiences with this kind of lag. In the next section, we look at how to power real-time experiences without any data or training.

Figure 14-2. Running a familiar photo on Google's Vision AI API to obtain object detection results

Figure 14-2. Running a familiar photo on Google's Vision AI API to obtain object detection results

Reusing a Pretrained Model

In this section, we explore how easy it is to get an object detector running on a mobile phone with a pretrained model. Hopefully, you are already comfortable with running a neural network on mobile from some of the previous chapters. The code to run object detection models on iOS and Android is referenced on the book's GitHub website (see *http://PracticalDeepLearning.ai*) at *code/chapter-14*. Changing functionality is simply a matter of replacing the existing *.tflite* model file with a new one.

In the previous chapters, we often used MobileNet to do classification tasks. Although the MobileNet family, including MobileNetV2 (2018) and MobileNetV3 (2019) by itself are classification networks only, they can act as a backbone for object detection architectures like SSD MobileNetV2, which we use in this chapter.

Reusing a Pretrained Model

In this section, we explore how easy it is to get an object detector running on a mobile phone with a pretrained model. Hopefully, you are already comfortable with running a neural network on mobile from some of the previous chapters. The code to run object detection models on iOS and Android is referenced on the book's GitHub website (see *http://PracticalDeepLearning.ai*) at *code/chapter-14*. Changing functionality is simply a matter of replacing the existing *.tflite* model file with a new one.

In the previous chapters, we often used MobileNet to do classification tasks. Although the MobileNet family, including MobileNetV2 (2018) and MobileNetV3 (2019) by itself are classification networks only, they can act as a backbone for object detection architectures like SSD MobileNetV2, which we use in this chapter.

Obtaining the Model

The best way to uncover the magic behind the scenes is to take a deeper look at a model. We'd need to get our hands on the TensorFlow Models repository, which features models for more than 50 deep learning tasks, including audio classification, text summarization, and object detection. This repository contains models as well as utility scripts that we use throughout this chapter. Without further ado, let's clone the repository on our machines:

```
$ git clone https://github.com/tensorflow/models.git && cd models/research
```

Then, we update the `PYTHONPATH` to make all scripts visible to Python:

```
$ export PYTHONPATH="${PYTHONPATH}:`pwd`:`pwd`/slim"
```

Next, we copy over the Protocol Buffer (protobuf) compiler command script from our GitHub repository (see *http://PracticalDeepLearning.ai*) into our current directory to make the `.proto` files available to Python:

```
$ cp {path_to_book_github_repo}/code/chapter-14/add_protoc.sh . && \
chmod +x add_protoc.sh && \
./add_protoc.sh
```

Finally, we run the *setup.py* script as follows:

```
$ python setup.py build
$ python setup.py install
```

Now is the time to download our prebuilt model. In our case, we're using the SSD MobileNetV2 model. You can find a large list of models at TensorFlow's object detection model zoo, featuring models by the datasets on which they were trained, speed of inference, and Mean Average Precision (mAP). Within that list, download the model titled `ssd_mobilenet_v2_coco`, which, as the name suggests, was trained on the MS COCO dataset. Benchmarked at 22 mAP, this model runs at 31 ms on an NVIDIA GeForce GTX TITAN X with a 600x600 resolution input. After downloading that model, we unzip it into the *models/research/object_detection* directory inside the TensorFlow repository. We

can inspect the contents of the directory as follows:

```
$ cd object_detection/
$ ls ssd_mobilenet_v2_coco_2018_03_29
checkpoint                      model.ckpt.data-00000-of-00001  model.ckpt.meta
saved_model
frozen_inference_graph.pb       model.ckpt.index                pipeline.config
```

can inspect the contents of the directory as follows:

```
$ cd object_detection/
$ ls ssd_mobilenet_v2_coco_2018_03_29
checkpoint                      model.ckpt.data-00000-of-00001  model.ckpt.meta
saved_model
frozen_inference_graph.pb       model.ckpt.index                pipeline.config
```

Test Driving Our Model

Before we plug our model into a mobile app, it's a good idea to verify whether the model is able to make predictions. The TensorFlow Models repository contains a Jupyter Notebook where we can simply plug in a photograph to make predictions on it. You can find the notebook at *models/research/object_detection/object_detection_tutorial.ipynb*. Figure 14-3 shows a prediction from that notebook on a familiar photograph.

Figure 14-3. Object detection prediction in the ready-to-use Jupyter Notebook from the TensorFlow Models repository

Test Driving Our Model

Before we plug our model into a mobile app, it's a good idea to verify whether the model is able to make predictions. The TensorFlow Models repository contains a Jupyter Notebook where we can simply plug in a photograph to make predictions on it. You can find the notebook at *models/research/object_detection/object_detection_tutorial.ipynb*. Figure 14-3 shows a prediction from that notebook on a familiar photograph.

Figure 14-3. Object detection prediction in the ready-to-use Jupyter Notebook from the TensorFlow Models repository

Deploying to a Device

Now that we've verified that the model works, it's time to convert it into a mobile-friendly format. For conversion to the TensorFlow Lite format, we use our familiar `tflite_convert` tool from Chapter 13. It should be noted that this tool operates on *.pb* files, whereas the TensorFlow Object Detection model zoo provides only model checkpoints. So, we first need to generate a *.pb* model from the checkpoints and graphs. Let's do that using the handy script that comes with the TensorFlow Models repository. You can find the *export_tflite_ssd_graph.py* file in *models/research/object_detection*:

```
$ python export_tflite_ssd_graph.py \
--pipeline_config_path=ssd_mobilenet_v2_coco_2018_03_29/pipeline.config \
--trained_checkpoint_prefix=ssd_mobilenet_v2_coco_2018_03_29/
model.ckpt.data-00000-of-00001 \
--output_directory=tflite_model \
--add_postprocessing_op=true
```

If the preceding script execution was successful, we'll see the following files in the *tflite_model* directory:

```
$ ls tflite_model
tflite_graph.pb
tflite_graph.pbtxt
```

We will convert this to the TensorFlow Lite format using the `tflite_convert` tool.

```
$ tflite_convert --graph_def_file=tflite_model/tflite_graph.pb \
--output_file=tflite_model/model.tflite
```

Now the only thing that remains is to plug the model into an app. The app that we'll be using is referenced in the book's GitHub website (see *http://PracticalDeepLearning.ai*) at *code/chapter-14*. We have already looked at how to swap a model in an app in Chapters 11, 12, and 13, so we won't discuss that here again. Figure 14-4 shows the result of the object detector model when plugged into the Android app.

Figure 14-4. Running a real-time object detector model on an Android device

The reason why we're able to see the "Cat" prediction right away is because 'Cat' is one of the 80 categories already present in the MS COCO dataset that was used to train the model. This model might be sufficient for Bob to deploy on his devices (keeping in mind that although Bob might not use mobile phones to track his garden, the process for deploying a TensorFlow Lite model to edge hardware is quite similar). However, Bob would do well for himself to improve the precision of his model by fine tuning it on data that has been generated within his garden. In the next section, we explore how to use transfer learning to build an object detector by using only a web-based tool. If you've read Chapter 8, what comes next should feel familiar.

Figure 14-4. Running a real-time object detector model on an Android device

The reason why we're able to see the "Cat" prediction right away is because 'Cat' is one of the 80 categories already present in the MS COCO dataset that was used to train the model. This model might be sufficient for Bob to deploy on his devices (keeping in mind that although Bob might not use mobile phones to track his garden, the process for deploying a TensorFlow Lite model to edge hardware is quite similar). However, Bob would do well for himself to improve the precision of his model by fine tuning it on data that has been generated within his garden. In the next section, we explore how to use transfer learning to build an object detector by using only a web-based tool. If you've read Chapter 8, what comes next should feel familiar.

Building a Custom Detector Without Any Code

My cat's breath smells like cat food!

—Ralph Wiggum

Let's face it. Your authors could go around and snap a lot of cat pictures within their gardens to perform this experiment. It would be tedious, and we might be rewarded with only a few extra percentage points of precision and recall. Or we could look at a really fun example that also tests the limits of CNNs. We are, of course, referring to *The Simpsons*. Given that most models are not trained with features from cartoon images, it would be interesting to see how our model performs here.

First, we need to get our hands on a dataset. Thankfully, we have one readily available on Kaggle. Let's download the dataset and use CustomVision.ai (similar to Chapter 8) to train our Simpsons Classifier using the following steps. *Excellent!*

NOTE

Contrary to what some may believe, we, the authors, do not have millions of dollars lying around. This means that we cannot afford to buy the rights to *The Simpsons* from Fox. Copyright laws, as a consequence, prevent us from publishing any images from that show. Instead, in this section, we are using the next best thing—public domain images from the United States Congress. In place of the images from the cartoon, we use the photographs of congresswomen and congressmen with the same first names as the Simpsons family: *Homer* Hoch, *Marge* Roukema, *Bart* Gordon, and *Lisa* Blunt Rochester. Keep in mind that this is only for publishing purposes; we have still trained on the original dataset from the cartoon.

1. Go to CustomVision.ai and start a new Object Detection Project (Figure 14-5).

Figure 14-5. Creating a new Object Detection project in CustomVision.ai

2. Create tags for Homer, Marge, Bart, and Lisa. Upload 15 images for each character and draw a bounding box for each of those images. The dashboard should resemble Figure 14-6 right about now.

Figure 14-6. Dashboard with bounding box and class name

3. Click the Train button and let the training commence. There are sliders to control the probability and overlap thresholds on the dashboard. We can play around with them and adjust them to something we like and with which get good precision and recall.

4. Click the Quick Test button and use any random image of the Simpsons (not previously used for training) to test the performance of the model. The results might not be great yet, but that's okay: we can fix it by training with more data.

5. Let's experiment with how many images we need to increase precision, recall, and mAP. Let's start by adding five more images for each class and retrain the model.

6. Repeat this process until we have acceptable results. Figure 14-7 shows the results from our experiment.

Figure 14-7. Measuring improvement in percent mean average precision with increasing number of images per class

With our final model, we are able to get reasonably good detections for a random image off the internet, as can be seen in Figure 14-8. Although not expected, it is surprising that a model pretrained on natural images is able to fine tune on cartoons with relatively little data.

Figure 14-8. Detected Simpsons characters with the final model, represented by US congress members of the same first name (see note at the beginning of this section)

As we already know by now, CustomVision.ai allows this model to be exported in a variety of formats including Core ML, TensorFlow Lite, ONNX, and more. We can simply download into our desired format (*.tflite* in our case) and plug this file into the app similar to see how we did with the pretrained model in the previous section. With real-time Simpsons detection in our hands, we could show it off to our friends and family the next time Principal Skinner talks about steamed hams on our TV.

NOTE

CustomVision.ai is not the only platform that allows you to label, train, and deploy online without writing any code. Google's Cloud AutoML (in beta as of October 2019) and Apple's Create ML (only for the Apple ecosystem) offer a very similar feature set. Additionally, Matroid allows you to build custom object detectors from video feeds, which can be extremely useful to train a network without spending too much effort on building a dataset.

So far, we have looked at three quick ways to get object detection running with a very minimal amount of code. We estimate that about 90% of use cases can be handled with one of these options. If you can solve for your scenario using these you can stop reading this chapter and jump straight to the "Case Studies" section.

In a very small set of scenarios, we might want further fine-grained control on factors such as accuracy, speech, model size, and resource usage. In the upcoming sections, we look at the world of object detection in a little more detail, from labeling the data all the way to deploying the model.

CustomVision.ai is not the only platform that allows you to label, train, and deploy online without writing any code. Google's Cloud AutoML (in beta as of October 2019) and Apple's Create ML (only for the Apple ecosystem) offer a very similar feature set. Additionally, Matroid allows you to build custom object detectors from video feeds, which can be extremely useful to train a network without spending too much effort on building a dataset.

So far, we have looked at three quick ways to get object detection running with a very minimal amount of code. We estimate that about 90% of use cases can be handled with one of these options. If you can solve for your scenario using these, you can stop reading this chapter and jump straight to the "Case Studies" section.

In a very small set of scenarios, we might want further fine-grained control on factors such as accuracy, speech, model size, and resource usage. In the upcoming sections, we look at the world of object detection in a little more detail, from labeling the data all the way to deploying the model.

The Evolution of Object Detection

Over the years, as the deep learning revolution happened, there was a renaissance not only for classification, but also for other computer-vision tasks, including object detection. During this time, several architectures were proposed for object detection, often building on top of each other. Figure 14-9 shows a timeline of some of them.

Figure 14-9. A timeline of different object detection architectures (image source: Recent Advances in Deep Learning for Object Detection by Xiongwei Wu et al.)

In object classification, our CNN architecture extracts the features from an image and computes the probabilities for a specific number of classes. These CNN architectures (ResNet, MobileNet) work as the *backbone* for object detection networks. In object detection networks, our end result is bounding boxes (defined by the center of rectangle, height, width). Although covering the inner details of many of these will probably need significantly more in-depth exploration (which you can find on this book's GitHub website, linked at *http://PracticalDeepLearning.ai*), we can broadly divide them into two categories, as shown in Table 14-3.

Table 14-3. Categories of object detectors

	Description	Pros and cons
Two-stage detectors	Generate category-agnostic bounding box candidates (regions of interest) using a Region Proposal Network (RPN). Run a CNN on these region proposals to assign each a	+ High Accuracy – Slower

	category. Examples: Faster R-CNN, Mask R-CNN.	
One-stage detectors	Directly make categorical predictions on objects on each location of the feature maps; can be trained end-to-end training. Examples: YOLO, Single-Shot Detector (SSD).	+ Speed, better suited for real-time applications – Lower accuracy

NOTE

In the world of object detection, Ross Girshik is a name you will encounter often while reading papers. Working before the deep learning era as well as during it, his work includes Deformable Part Models (DPM), R-CNN, Fast R-CNN, Faster R-CNN, You Only Look Once (YOLO), Mask R-CNN, Feature Pyramid Network (FPN), RetinaNet, and ResNeXt to name just a few, often breaking his own records year after year on public benchmarks.

> *I participated in several first-place entries into the PASCAL VOC object detection challenge, and was awarded a "lifetime achievement" prize for my work on deformable part models. I think this refers to the lifetime of the PASCAL challenge—and not mine!*
>
> *—Ross Girschik*

	category. Examples: Faster R-CNN, Mask R-CNN.	
One-stage detectors	Directly make categorical predictions on objects on each location of the feature maps; can be trained end-to-end training. Examples: YOLO, Single-Shot Detector (SSD).	+ Speed, better suited for real-time applications – Lower accuracy

NOTE

In the world of object detection, Ross Girshik is a name you will encounter often while reading papers. Working before the deep learning era as well as during it, his work includes Deformable Part Models (DPM), R-CNN, Fast R-CNN, Faster R-CNN, You Only Look Once (YOLO), Mask R-CNN, Feature Pyramid Network (FPN), RetinaNet, and ResNeXt to name just a few, often breaking his own records year after year on public benchmarks.

> *I participated in several first-place entries into the PASCAL VOC object detection challenge, and was awarded a "lifetime achievement" prize for my work on deformable part models. I think this refers to the lifetime of the PASCAL challenge—and not mine!*
>
> *—Ross Girschik*

Performance Considerations

From a practitioner's point of view, the primary concerns tend to be accuracy and speed. These two factors usually have an inverse relationship with each other. We want to find the detector that strikes the ideal balance for our scenario. Table 14-4 summarizes some of those readily available pretrained models on the TensorFlow object detection model zoo. The speed reported was on a 600x600 pixel resolution image input, processed on an NVIDIA GeForce GTX TITAN X card.

Table 14-4. Speed versus percent mean average precision for some readily available architectures on the TensorFlow object detection model zoo website

	Inference speed (ms)	% mAP on MS COCO
ssd_mobilenet_v1_coco	30	21
ssd_mobilenet_v2_coco	31	22
ssdlite_mobilenet_v2_coco	27	22
ssd_resnet_50_fpn_coco	76	35
faster_rcnn_nas	1,833	43

A 2017 study done by Google (see Figure 14-10) revealed the following key findings:

- One-stage detectors are faster than two-stage, though less accurate.
- The higher the accuracy of the backbone CNN architecture on classificatior tasks (such as ImageNet), the higher the precision of the object detector.
- Small-sized objects are challenging for object detectors, which give poor performance (usually under 10% mAP). The effect is more severe on one-stage detectors.
- On large-sized objects, most detectors give similar performance.
- Higher-resolution images give better results because small objects appear larger to the network.

- Two-stage object detectors internally generate a large number of proposals to be considered as the potential locations of objects. The number of proposals can be tuned. Decreasing the number of proposals can lead to speedups with little loss in accuracy (the threshold will depend on the use case).

Figure 14-10. The effect of object detection architecture as well as the backbone architecture (feature extractor) on the percent mean average precision and prediction time (note that the colors might not be clear in grayscale printing; refer to the book's GitHub website [see http://PracticalDeepLearning.ai] for the color image) (image source: Speed/accuracy trade-offs for modern convolutional object detectors by Jonathan Huang et al.)

1

- Two-stage object detectors internally generate a large number of proposals to be considered as the potential locations of objects. The number of proposals can be tuned. Decreasing the number of proposals can lead to speedups with little loss in accuracy (the threshold will depend on the use case).

Figure 14-10. The effect of object detection architecture as well as the backbone architecture (feature extractor) on the percent mean average precision and prediction time (note that the colors might not be clear in grayscale printing; refer to the book's GitHub website [see http://PracticalDeepLearning.ai] for the color image) (image source: Speed/accuracy trade-offs for modern convolutional object detectors by Jonathan Huang et al.)

Key Terms in Object Detection

The terms used heavily in object detection include Intersection over Union, Mean Average Precision, Non-Maximum Suppression, and Anchor. Let's examine what each of these means.

Key Terms in Object Detection

The terms used heavily in object detection include Intersection over Union, Mean Average Precision, Non-Maximum Suppression, and Anchor. Let's examine what each of these means.

Intersection over Union

In an object detection training pipeline, metrics such as Intersection over Union (IoU) are typically used as a threshold value to filter detections of a certain quality. To calculate the IoU, we use both the predicted bounding box and the ground truth bounding box to calculate two different numbers. The first is the area where both the prediction and the ground truth overlap—the intersection. The second is the span covered by both the prediction and the ground truth—the union. As the name suggests, we simply divide the total area of the intersection by the area of the union to get the IoU. Figure 14-11 visually demonstrates the IoU concept for two 2x2 squares that intersect in a single 1x1 square in the middle.

Figure 14-11. A visual representation of the IoU ratio

In the perfect scenario in which the predicted bounding box matches the ground truth exactly, the IoU value would be 1. In the worst-case scenario, the prediction would have no overlap with the ground truth and the IoU value would be 0. As we can see, the IoU value would range between 0 and 1, with higher numbers indicating higher-quality predictions, as illustrated in Figure 14-12.

To help us filter and report results, we would set a minimum IoU threshold value. Setting the threshold to a really aggressive value (such as 0.9) would result in a loss of a lot of predictions that might turn out to be important further down the pipeline. Conversely, setting the threshold value too low would result in too many spurious bounding boxes. A minimum IoU of 0.5 is typically used for reporting the precision of an object detection model.

Figure 14-12. IoU illustrated; predictions from better models tend to have heavier overlap with the ground truth, resulting in a higher IoU

It’s worth reiterating here that the IoU value is calculated per instance of a category, not per image. However, to calculate the quality of the detector over a larger set of images, we would use IoU in conjunction with other metrics such as Average Precision.

To help us filter and report results, we would set a minimum IoU threshold value. Setting the threshold to a really aggressive value (such as 0.9) would result in a loss of a lot of predictions that might turn out to be important further down the pipeline. Conversely, setting the threshold value too low would result in too many spurious bounding boxes. A minimum IoU of 0.5 is typically used for reporting the precision of an object detection model.

Figure 14-12. IoU illustrated; predictions from better models tend to have heavier overlap with the ground truth, resulting in a higher IoU

It’s worth reiterating here that the IoU value is calculated per instance of a category, not per image. However, to calculate the quality of the detector over a larger set of images, we would use IoU in conjunction with other metrics such as Average Precision.

Mean Average Precision

While reading research papers on object detection, we often come across numbers such as AP@0.5. That term conveys the average precision at IoU=0.5. Another more elaborate representation would be AP@[0.6:0.05:0.75], which is the average precision from IoU=0.6 to IoU=0.75 at increments of 0.05. The Mean Average Precision (or mAP) is simply the average precision across all categories. For the COCO challenge, the MAP metric used is AP@[0.5:0.05:0.95].

Mean Average Precision

While reading research papers on object detection, we often come across numbers such as AP@0.5. That term conveys the average precision at IoU=0.5. Another more elaborate representation would be AP@[0.6:0.05:0.75], which is the average precision from IoU=0.6 to IoU=0.75 at increments of 0.05. The Mean Average Precision (or mAP) is simply the average precision across all categories. For the COCO challenge, the MAP metric used is AP@[0.5:0.05:0.95].

Non-Maximum Suppression

Internally, object detection algorithms make a number of proposals for the potential locations of objects that might be in the image. For each object in the image, it is expected to have multiple of these bounding box proposals at varying confidence values. Our task is to find the one proposal that best represents the real location of the object. A naïve way would be to consider only the proposal with the maximum confidence value. This approach might work if there were only one object in the entire image. But this won't work if there are multiple categories, each with multiple instances in a single image.

Non-Maximum Suppression (NMS) comes to the rescue (Figure 14-13). The key idea behind NMS is that two instances of the same object will not have heavily overlapping bounding boxes; for instance, the IoU of their bounding boxes will be less than a certain IoU threshold (say 0.5). A greedy way to achieve this is by repeating the following steps for each category:

1. Filter out all proposals under a minimum confidence threshold.
2. Accept the proposal with the maximum confidence value.
3. For all remaining proposals sorted in descending order of their confidence values, check whether the IoU of the current box with respect to one of the previously accepted proposals ≥ 0.5. If so, filter it; otherwise, accept it as a proposal.

Figure 14-13. Using NMS to find the bounding box that best represents the location of the object in an image

ʒ

/

Using the TensorFlow Object Detection API to Build Custom Models

In this section, we run through a full end-to-end example of building an object detector. We look at several steps in the process, including collecting, labeling, and preprocessing data, training the model, and exporting it to the TensorFlow Lite format.

First, let's look at some strategies for collecting data.

Using the TensorFlow Object Detection API to Build Custom Models

In this section, we run through a full end-to-end example of building an object detector. We look at several steps in the process, including collecting, labeling, and preprocessing data, training the model, and exporting it to the TensorFlow Lite format.

First, let’s look at some strategies for collecting data.

Data Collection

We know by now that, in the world of deep learning, data reigns supreme. There are a few ways in which we can acquire data for the object that we want to detect:

Using ready-to-use datasets

There are a few public datasets available to train object detection models such as MS COCO (80 categories), ImageNet (200 categories), Pascal VOC (20 categories), and the newer Open Images (600 categories). MS COCO and Pascal VOC are used in most object detection benchmarks, with COCO-based benchmarks being more realistic for real-world scenarios due to more complex imagery.

Downloading from the web

We should already be familiar with this process from Chapter 12 in which we collected images for the “Hot Dog” and “Not Hot Dog” classes. Browser extensions such as Fatkun come in handy for quickly downloading a large number of images from search engine results.

Taking pictures manually

This is the more time consuming (but potentially fun) option. For a robust model, it’s important to train it with pictures taken in a variety of different settings. With the object to detect in hand, we should take at least 100 to 150 images of it against different backgrounds, with a variety of compositions (framing) and angles, and in many different lighting conditions. Figure 14-14 shows some examples of pictures taken to train a Coke and Pepsi object detection model. Considering that a model can potentially learn the spurious correlation that red means Coke and blue means Pepsi, it’s a good idea to mix objects with backgrounds that could potentially confuse it so that eventually we realize a more robust model.

Figure 14-14. Photographs of objects taken in a variety of different settings to train an object detector model

It's worth taking pictures of the object in environments that it's unlikely to be used in order to diversify the dataset and improve the robustness of the model. It also has the added bonus of keeping an otherwise boring and tedious process a little entertaining. We can make it interesting by challenging ourselves to come up with unique and innovative photos. Figure 14-15 shows some creative examples of photos taken during the process of building a currency detector.

Figure 14-15. Some creative photographs taken during the process of building a diverse currency dataset

Because we want to make a cat detector, we will be reusing the cat images from the “Cats and Dogs” Kaggle dataset that we used in Chapter 3. We randomly chose images from that set and split them into a train and test set.

To maintain uniformity of input size to our network, we resize all images to a fixed size. We will use the same size as part of our network definition and while converting to a *.tflite* model. We can use the ImageMagick tool to resize all images at once:

```
$ apt-get install imagemagick
$ mogrify -resize 800x600 *.jpg
```

TIP

If you have a large number of images (i.e., several tens of thousands of images) in the current directory, the preceding command might fail to enlist all of the images. A solution would be to use the `find` command to list all of the images and pipe the output to the `mogrify` command:

```
$ find . -type f | awk -F. '!a[$NF]++{print $NF}' |
xargs -I{} mogrify -resize 800x600 *.jpg
```

Labeling the Data

After we have our data, the next step would be to label it. Unlike classification where simply placing an image in the correct directory would be sufficient, we need to manually draw the bounding rectangles for the objects of interest. Our tool of choice for this task is LabelImg (available for Windows, Linux, and Mac) for the following reasons:

- You can save annotations either as XML files in PASCAL VOC format (also adopted by ImageNet) or the YOLO format.
- It supports single and multiclass labels.

NOTE

If you are the curious cat like us, you'd want to know what the difference is between the YOLO format and the PASCAL VOC format. YOLO happens to use simple *.txt* files, one per image with the same name, to store information about the classes and the bounding boxes within the image. The following is what a *.txt* file in YOLO format typically looks like:

```
class_for_box1 box1_x box1_y box1_w box1_h
class_for_box2 box2_x box2_y box2_w box2_h
```

Note that the x, y, width, and height in this example are normalized using the full width and height of the image.

PASCAL VOC format, on the other hand, is XML based. Similar to the YOLO format, we use one XML file per image (ideally with the same name). You can view the sample format on the Git repository at *code/chapter-14/pascal-voc-sample.xml*.

First, download the application from GitHub to a directory on your computer. This directory will have an executable and a data directory containing some sample data. Because we will be using our own custom data, we don't need the data within the provided data directory. You can start the application by double-clicking the executable file.

At this point, we should have a collection of images that is ready to be used during the training process. We first divide our data randomly into training and test sets and place them in separate directories. We can do this split by simply

dragging and dropping the images randomly into either directory. After the train and test directories are created, we load the training directory by clicking Open Dir, as shown in Figure 14-16.

Figure 14-16. Click the Open Dir button and then select the directory that contains the training data

After LabelImg loads the training directory, we can start our labeling process. We must go through each image and manually draw bounding boxes for each object (only cats in our case), as shown in Figure 14-17. After we make the bounding box, we are prompted to provide a label to accompany the bounding box. For this exercise, enter the name of the object, “cat”. After a label is entered, we need only to select the checkbox to specify that label again for subsequent images. For images with multiple objects, we would make multiple bounding boxes and add their corresponding labels. If there are different kinds o objects, we simply add a new label for that object class.

Figure 14-17. Select the Create RectBox from the panel on the left to make a bounding box that covers the cat

Now, repeat this step for all of the training and test images. We want to obtain tight bounding boxes for each object, such that all parts of the object are bounded by the box. Finally, in the train and test directories, we see *.xml* files accompanying each image file, as depicted in Figure 14-18. We can open the `.xml` file in a text editor and inspect metadata such as image file name, bounding box coordinates, and label name.

f

Figure 14-18. Each image is accompanied by an XML file that contains the label information and the bounding box information

In Chapter 1, we mentioned that during the development of the MS COCO dataset, it took roughly three seconds to label the name of each object in an image, approximately 30 seconds to place a bounding box around each object, and 79 seconds to draw the outlines for each object.

Labeling for detection can take about 10x the time as labeling for classification. Doing this on a large scale while maintaining high quality can get really expensive, really quickly. What's the expensive part here? Drawing the bounding boxes manually. What is something cheaper than drawing them? Verifying the correctness of already drawn boxes. And that's what we want to maximize.

Rather than labeling large quantities of data manually, we'd go through an iterative semiautomated process of labeling. In this approach you'd do the following:

1. Pick a small fraction of images from the pool of data.
2. Manually label it with the object class and bounding box information.

3. Train a model with the labeled data.
4. Make predictions on the remaining unlabeled data using this model.
5. If a prediction confidence is higher than a set threshold, assign that as the label.
6. Pick a sample of the remaining predictions under this threshold.
7. In this sample, verify the predictions for correctness. If a prediction is incorrect, manually fix it. In practice, this usually means moving the bounding box slightly, which is much quicker than drawing the full bounding boxes.
8. Add the images from the manually reviewed sample to the training dataset.
9. Repeat the process until the model achieves acceptable performance on a separate validation set.

In the first few iterations of this process, you might have a barely functioning model. But that's okay. By using this human-in-the-loop approach, we are constantly teaching it what it's bad at. Pretty soon, our model will become confident enough to automatically label a majority of the data, cutting down on the large costs of manually labeling large amounts of data.

This approach is one way of using the technique known as *active learning*. Famous labeling companies like Figure Eight use this heavily to reduce labeling costs and increase efficiency so that a large majority of time is eventually spent on verification.

3. Train a model with the labeled data.
4. Make predictions on the remaining unlabeled data using this model.
5. If a prediction confidence is higher than a set threshold, assign that as the label.
6. Pick a sample of the remaining predictions under this threshold.
7. In this sample, verify the predictions for correctness. If a prediction is incorrect, manually fix it. In practice, this usually means moving the bounding box slightly, which is much quicker than drawing the full bounding boxes.
8. Add the images from the manually reviewed sample to the training dataset.
9. Repeat the process until the model achieves acceptable performance on a separate validation set.

In the first few iterations of this process, you might have a barely functioning model. But that's okay. By using this human-in-the-loop approach, we are constantly teaching it what it's bad at. Pretty soon, our model will become confident enough to automatically label a majority of the data, cutting down on the large costs of manually labeling large amounts of data.

This approach is one way of using the technique known as *active learning*. Famous labeling companies like Figure Eight use this heavily to reduce labeling costs and increase efficiency so that a large majority of time is eventually spent on verification.

Preprocessing the Data

At this point, we have handy XML data that gives the bounding boxes for all of our objects. However, for us to use TensorFlow for training on the data, we must preprocess it into a format that TensorFlow understands, namely TFRecords. Before we can convert our data into TFRecords, we must go through the intermediate step of consolidating the data from all the XML files into a single comma-separated values (CSV) file. TensorFlow provides helper scripts that assist us with these operations.

Now that we have our environment set up, it's time to begin doing some real work:

1. Convert our cats dataset directory to a single CSV file using the `xml_to_csv` tool from the racoon_dataset repository by Dat Tran. We have provided a slightly edited copy of this file in our repository at *code/chapter-14/xml_to_csv.py*:

```
$ python xml_to_csv.py -i {path to cats training dataset} -o {path to output train_labels.csv}
```

2. Do the same for the test data:

```
$ python xml_to_csv.py -i {path to cats test dataset} -o {path to test_labels.csv}
```

3. Make the *label_map.pbtxt* file. This file contains the label and identifier mappings for all of our classes. We use this file to convert our text label into an integer identifier, which is what TFRecord expects. Because we have only one class, the file is rather small and looks as follows:

```
item {
    id: 1
    name: 'cat'
}
```

4. Generate the TFRecord format files that will contain the data to be used for training and testing our model later on. This file is also from the racoon_dataset repository by Dat Tran. We have provided a slightly edited copy of this file in our repository at *code/chapter-14/generate_tfrecord.py*. (It's worth noting that the `image_dir` path in the argument should be the same as the path in the XML files; LabelImg uses absolute paths.)

```
$ python generate_tfrecord.py \
--csv_input={path to train_labels.csv} \
--output_path={path to train.tfrecord} \
--image_dir={path to cat training dataset}
```

```
$ python generate_tfrecord.py \
--csv_input={path to test_labels.csv} \
--output_path={path to test.tfrecord} \
--image_dir={path to cat test dataset}
```

With our *train.tfrecord* and *test.tfrecord* files ready, we can now begin our training process.

4. Generate the TFRecord format files that will contain the data to be used for training and testing our model later on. This file is also from the racoon_dataset repository by Dat Tran. We have provided a slightly edited copy of this file in our repository at *code/chapter-14/generate_tfrecord.py*. (It's worth noting that the `image_dir` path in the argument should be the same as the path in the XML files; LabelImg uses absolute paths.)

```
$ python generate_tfrecord.py \
--csv_input={path to train_labels.csv} \
--output_path={path to train.tfrecord} \
--image_dir={path to cat training dataset}
```

```
$ python generate_tfrecord.py \
--csv_input={path to test_labels.csv} \
--output_path={path to test.tfrecord} \
--image_dir={path to cat test dataset}
```

With our *train.tfrecord* and *test.tfrecord* files ready, we can now begin our training process.

Inspecting the Model

(This section is informational only and is not necessary for the training process. You can skip ahead to "Training" if you want to.)

We can inspect our model using the `saved_model_cli` tool:

```
$ saved_model_cli show --dir ssd_mobilenet_v2_coco_2018_03_29/saved_model \
--tag_set serve \
--signature_def serving_default
```

The output of that script looks like the following:

```
The given SavedModel SignatureDef contains the following input(s):
  inputs['inputs'] tensor_info:
      dtype: DT_UINT8
      shape: (-1, -1, -1, 3)
      name: image_tensor:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['detection_boxes'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 100, 4)
      name: detection_boxes:0
  outputs['detection_classes'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 100)
      name: detection_classes:0
  outputs['detection_scores'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 100)
      name: detection_scores:0
  outputs['num_detections'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1)
      name: num_detections:0
  outputs['raw_detection_boxes'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, -1, 4)
      name: raw_detection_boxes:0
  outputs['raw_detection_scores'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, -1, 2)
      name: raw_detection_scores:0
Method name is: tensorflow/serving/predict
```

The following is how we can interpret the output from the previous command:

1. The shape of `inputs` being `(-1, -1, -1, 3)` indicates that the trained model can accept any arbitrarily sized image input that contains three channels. Due to this inherent flexibility in input size, the converted model would be larger than if the model had a fixed-size input. When we train our own object detector later in this chapter, we will be fixing the input size to 800 x 600 pixels to make the resulting model more compact.

2. Moving on to the outputs, the very first output is `detection_boxes (-1, 100, 4)`. This tells us how many detection boxes are possible, and what will they look like. In particular, the very first number (i.e., -1) indicates that we can have any number of boxes based on the detections for all 100 classes (second number) with four coordinates (third number)—x, y, width, and height—defining each box. In other words, we are not limiting the number of detections for each of the 100 classes.

3. For `detection_classes`, we have a list of two elements: the first defining how many objects we detected, and the second, a one-hot encoded vector with the detected class enabled.

4. `num_detections` is the number of objects detected in the image. It is a single floating point.

5. `raw_detection_boxes` defines the coordinates of each box that was detected for each object. All of these detections are before the application of NMS on all the detections.

6. `raw_detection_scores` is a list of two floating-point numbers, the first describing the total number of detections, and the second giving the total number of categories including the background if it is considered a separate category.

The following is how we can interpret the output from the previous command:

1. The shape of `inputs` being `(-1, -1, -1, 3)` indicates that the trained model can accept any arbitrarily sized image input that contains three channels. Due to this inherent flexibility in input size, the converted model would be larger than if the model had a fixed-size input. When we train our own object detector later in this chapter, we will be fixing the input size to 800 x 600 pixels to make the resulting model more compact.

2. Moving on to the outputs, the very first output is `detection_boxes (-1, 100, 4)`. This tells us how many detection boxes are possible, and what will they look like. In particular, the very first number (i.e., -1) indicates that we can have any number of boxes based on the detections for all 100 classes (second number) with four coordinates (third number)—x, y, width, and height—defining each box. In other words, we are not limiting the number of detections for each of the 100 classes.

3. For `detection_classes`, we have a list of two elements: the first defining how many objects we detected, and the second, a one-hot encoded vector with the detected class enabled.

4. `num_detections` is the number of objects detected in the image. It is a single floating point.

5. `raw_detection_boxes` defines the coordinates of each box that was detected for each object. All of these detections are before the application of NMS on all the detections.

6. `raw_detection_scores` is a list of two floating-point numbers, the first describing the total number of detections, and the second giving the total number of categories including the background if it is considered a separate category.

Training

Because we are using SSD MobileNetV2, we will need to create a copy of the *pipeline.config* file (from the TensorFlow repository) and modify it based on our configuration parameters. First, make a copy of the configuration file and search for the string `PATH_TO_BE_CONFIGURED` within the file. This parameter indicates all of the places that need to be updated with the correct paths (absolute paths preferably) within our filesystem. We will also want to edit some parameters in the configuration file, such as number of classes (`num_classes`), number of steps (`num_steps`), number of validation samples (`num_examples`), and path of label mappings for both the train and test/eval sections. (You'll find our version of the *pipeline.config* file on the book's GitHub website (see *http://PracticalDeepLearning.ai*).

```
$ cp object_detection/samples/configs/ssd_mobilenet_v2_coco.config
./pipeline.config
$ vim pipeline.config
```

NOTE

In our example, we're using the SSD MobileNetV2 model to train our object detector. Under different conditions, you might want to choose a different model on which to base your training. Because each model comes with its own pipeline configuration file, you'd be modifying that configuration file, instead. The configuration file comes bundled with each model. You'd use a similar process in which you'd identify the paths that need to be changed and update them using a text editor accordingly.

In addition to modifying paths, you might want to modify other parameters within the configuration file such as image size, number of classes, optimizer, learning rate, and number of epochs.

So far, we've been at the *models/research* directory within the TensorFlow repository. Let's now move into the *object_detection* directory and run the *model_main.py* script from TensorFlow, which trains our model based on the configuration provided by the *pipeline.config* file that we just edited:

```
$ cd object_detection/
$ python model_main.py \
--pipeline_config_path=../pipeline.config \
--logtostderr \
```

```
--model_dir=training/
```

We'll know the training is happening correctly when we see lines of output that look like the following:

```
Average Precision (AP) @[ IoU=0.50:0.95 | area=all | maxDets=100 ] = 0.760
```

Depending on the number of iterations that we are training for, the training will take from a few minutes to a couple of hours, so plan to have a snack, do the laundry, clean the litter box, or better yet, read the chapter ahead. After the training is complete, we should see the latest checkpoint file in the training/ directory. The following files are also created as a result of the training process:

```
$ ls training/

checkpoint                                      model.ckpt-13960.meta
eval_0                                          model.ckpt-16747.data-00000-of-00001
events.out.tfevents.1554684330.computername     model.ckpt-16747.index
events.out.tfevents.1554684359.computername     model.ckpt-16747.meta
export                                          model.ckpt-19526.data-00000-of-00001
graph.pbtxt                                   model.ckpt-19526.index
label_map.pbtxt                               model.ckpt-19526.meta
model.ckpt-11180.data-00000-of-00001          model.ckpt-20000.data-00000-of-00001
model.ckpt-11180.index                        model.ckpt-20000.index
model.ckpt-11180.meta                         model.ckpt-20000.meta
model.ckpt-13960.data-00000-of-00001          pipeline.config
model.ckpt-13960.index
```

We convert the latest checkpoint file to the *.TFLite* format in the next section.

TIP

Sometimes, you might want to do a deeper inspection of your model for debugging, optimization, and/or informational purposes. Think of it like a DVD extra of a behind-the-scenes shot of your favorite TV show. You would run the following command along with arguments pointing to the checkpoint file, the configuration file, and the input type to view all the different parameters, the model analysis report, and other golden nuggets of information:

```
$ python export_inference_graph.py \
--input_type=image_tensor \
--pipeline_config_path=training/pipeline.config \
--output_directory=inference_graph \
--trained_checkpoint_prefix=training/model.ckpt-20000
```

The output from the this command looks like the following:

```
=========================Options=========================
-max_depth                  10000
-step                       -1
...

==================Model Analysis Report==================
...
Doc:
scope: The nodes in the model graph are organized by
their names, which is hierarchical like filesystem.
param: Number of parameters (in the Variable).

Profile:
node name | # parameters
_TFProfRoot (--/3.72m params)
 BoxPredictor_0 (--/93.33k params)
    BoxPredictor_0/BoxEncodingPredictor
    (--/62.22k params)
      BoxPredictor_0/BoxEncodingPredictor/biases
      (12, 12/12 params)
 ...
FeatureExtractor (--/2.84m params)
...
```

This report helps you to learn how many parameters you have, which could in turn help you in finding opportunities for model optimization.

The output from the this command looks like the following:

```
=========================Options=========================
-max_depth                  10000
-step                       -1
...

==================Model Analysis Report==================
...
Doc:
scope: The nodes in the model graph are organized by
their names, which is hierarchical like filesystem.
param: Number of parameters (in the Variable).

Profile:
node name | # parameters
_TFProfRoot (--/3.72m params)
 BoxPredictor_0 (--/93.33k params)
    BoxPredictor_0/BoxEncodingPredictor
    (--/62.22k params)
      BoxPredictor_0/BoxEncodingPredictor/biases
      (12, 12/12 params)
 ...
FeatureExtractor (--/2.84m params)
...
```

This report helps you to learn how many parameters you have, which could in turn help you in finding opportunities for model optimization.

Model Conversion

Now that we have the latest checkpoint file (the suffix should match the number of epochs in the *pipeline.config* file), we will feed it into the `export_tflite_ssd_graph` script like we did earlier in the chapter with our pretrained model:

```
$ python export_tflite_ssd_graph.py \
--pipeline_config_path=training/pipeline.config \
--trained_checkpoint_prefix=training/model.ckpt-20000 \
--output_directory=tflite_model \
--add_postprocessing_op=true
```

If the preceding script execution was successful, we'll see the following files in the *tflite_model* directory:

```
$ ls tflite_model
tflite_graph.pb
tflite_graph.pbtxt
```

We have one last step remaining: convert the frozen graph files to the *.TFLite* format. We can do that using the `tflite_convert` tool:

```
$ tflite_convert --graph_def_file=tflite_model/tflite_graph.pb \
--output_file=tflite_model/cats.tflite \
--input_arrays=normalized_input_image_tensor \
--
output_arrays='TFLite_Detection_PostProcess','TFLite_Detection_PostProcess:1','TFLi
te_Detection_PostProcess:2','TFLite_Detection_PostProcess:3' \
--input_shape=1,800,600,3 \
--allow_custom_ops
```

In this command, notice that we've used a few different arguments that might not be intuitive at first sight:

- For `--input_arrays`, the argument simply indicates that the images that will be provided during predictions will be normalized `float32` tensors.
- The arguments supplied to `--output_arrays` indicate that there will be four different types of information in each prediction: the number of

bounding boxes, detection scores, detection classes, and the coordinates of bounding boxes themselves. This is possible because we used the argument `--add_postprocessing_op=true` in the export graph script in the previous step.

- For `--input_shape`, we provide the same dimension values as we did in the *pipeline.config* file.

The rest are fairly trivial. We should now have a spanking new *cats.tflite* model file within our *tflite_model/* directory. It is now ready to be plugged into an Android, iOS, or edge device to do live detection of cats. This model is well on its way to saving Bob's garden!

TIP

In the floating-point MobileNet model, '`normalized_image_tensor`' has values between `[-1,1)`. This typically means mapping each pixel (linearly) to a value between [–1,1]. Input image values between 0 and 255 are scaled by (1/128) and then a value of –1 is added to them to ensure the range is `[-1,1)`.

In the quantized MobileNet model, '`normalized_image_tensor`' has values between `[0, 255]`.

In general, see the `preprocess` function defined in the feature extractor class in the TensorFlow Models repository at *models/research/object_detection/models* directory.

bounding boxes, detection scores, detection classes, and the coordinates of bounding boxes themselves. This is possible because we used the argument `--add_postprocessing_op=true` in the export graph script in the previous step.

- For `--input_shape`, we provide the same dimension values as we did in the *pipeline.config* file.

The rest are fairly trivial. We should now have a spanking new *cats.tflite* model file within our *tflite_model/* directory. It is now ready to be plugged into an Android, iOS, or edge device to do live detection of cats. This model is well on its way to saving Bob's garden!

TIP

In the floating-point MobileNet model, '`normalized_image_tensor`' has values between `[-1,1)`. This typically means mapping each pixel (linearly) to a value between [–1,1]. Input image values between 0 and 255 are scaled by (1/128) and then a value of –1 is added to them to ensure the range is `[-1,1)`.

In the quantized MobileNet model, '`normalized_image_tensor`' has values between `[0, 255]`.

In general, see the `preprocess` function defined in the feature extractor class in the TensorFlow Models repository at *models/research/object_detection/models* directory.

Image Segmentation

Taking a step further beyond object detection, to get more precise locations of objects, we can perform object segmentation. As we saw earlier in this chapter, this involves making category predictions for each pixel in an image frame. Architectures such as U-Net, Mask R-CNN, and DeepLabV3+ are commonly used to perform segmentation tasks. Similar to object detection, there has been a growing trend of running segmentation networks in real time, including on resource-constrained devices such as smartphones. Being real time opens up many consumer app scenarios like face filters (Figure 14-19), and industrial scenarios such as detecting drivable roads for autonomous cars (Figure 14-20).

Figure 14-19. Colorizing hair with ModiFace app by accurately mapping the pixels belonging to the hair

Figure 14-20. Image segmentation performed on frames from a dashcam (CamVid dataset)

As you might have grown accustomed to hearing in this book by now, you can accomplish much of this without needing much code. Labeling tools like Supervisely, LabelBox, and Diffgram can help not just label the data, but also load previously annotated data and train further on a pretrained object

segmentation model. Moreover, these tools provide AI-assisted labeling, which can dramatically speed up (~10 times) the otherwise laborious and expensive labeling process. If this sounds exciting, you are in luck! We have included a bonus resource guide on how to learn and build segmentation projects on the book's GitHub website (see *http://PracticalDeepLearning.ai*).

segmentation model. Moreover, these tools provide AI-assisted labeling, which can dramatically speed up (~10 times) the otherwise laborious and expensive labeling process. If this sounds exciting, you are in luck! We have included a bonus resource guide on how to learn and build segmentation projects on the book's GitHub website (see *http://PracticalDeepLearning.ai*).

Case Studies

Let’s now look at how object detection is being used to power real-world applications in the industry.

Case Studies

Let’s now look at how object detection is being used to power real-world applications in the industry.

Smart Refrigerator

Smart fridges have been gaining popularity for the past few years because they have started to become more affordable. People seem to like the convenience of knowing what's in their fridge even when they are not home to look inside. Microsoft teamed up with Swiss manufacturing giant Liebherr to use deep learning in its new generation SmartDeviceBox refrigerator (Figure 14-21). The company used Fast R-CNN to perform object detection within its refrigerators to detect different food items, maintain an inventory, and keep the user abreast of the latest state.

Figure 14-21. Detected objects along with their classes from the SmartDeviceBox refrigerator (image source)

Smart Refrigerator

Smart fridges have been gaining popularity for the past few years because they have started to become more affordable. People seem to like the convenience of knowing what's in their fridge even when they are not home to look inside. Microsoft teamed up with Swiss manufacturing giant Liebherr to use deep learning in its new generation SmartDeviceBox refrigerator (Figure 14-21). The company used Fast R-CNN to perform object detection within its refrigerators to detect different food items, maintain an inventory, and keep the user abreast of the latest state.

Figure 14-21. Detected objects along with their classes from the SmartDeviceBox refrigerator (image source)

Crowd Counting

Counting people is not just something that only cats do while staring out of windows. It's useful in a lot of situations including managing security and logistics at large sporting events, political gatherings, and other high-traffic areas. Crowd counting, as its name suggests, can be used to count any kind of object including people and animals. Wildlife conservation is an exceptionally challenging problem because of the lack of labeled data and intense data collection strategies.

Wildlife conservation

Several organizations, including the University of Glasgow, University of Cape Town, Field Museum of Natural History, and Tanzania Wildlife Research Institute, came together to count the largest terrestrial animal migration on earth: 1.3 million blue wildebeest and 250,000 zebra between the Serengeti and the Masaai Mara National Reserve, Kenya (Figure 14-22). They employed data labeling through 2,200 volunteer citizen scientists using a platform called Zooniverse and used automated object detection algorithms like YOLO for counting the wildebeest and zebra population. They observed that the count from the volunteers and from the object detection algorithms were within 1% of each other.

Figure 14-22. An aerial photograph of wildebeest taken from a small survey airplane (image source)

Kumbh Mela

Another example of dense crowd counting is from Prayagraj, India where approximately 250 million people visit the city once every 12 years for the Kumbh Mela festival (Figure 14-23). Crowd control for such large crowds has historically been difficult. In fact, in the year 2013, due to poor crowd management, a stampede broke out tragically resulting in the loss of 42 lives.

In 2019, the local state government contracted with Larsen & Toubro to use AI for a variety of logistical tasks including traffic monitoring, garbage collection, security assessments, and also crowd monitoring. Using more than a thousand CCTV cameras, authorities analyzed crowd density and designed an alerting system based on the density of people in a fixed area. In places of higher density the monitoring was more intense. It made the Guinness Book of World Records for being the largest crowd management system ever built in the history of humankind!

Figure 14-23. The 2013 Kumbh Mela, as captured by an attendee (image source)

;

Figure 14-23. The 2013 Kumbh Mela, as captured by an attendee (image source)

Face Detection in Seeing AI

Microsoft's Seeing AI app (for the blind and low-vision community) provides a real-time facial detection feature with which it informs the user of the people in front of the phone's camera, their relative locations, and their distance from the camera. A typical guidance might sound like "One face in the top-left corner, four feet away." Additionally, the app uses the facial-detection guidance to perform facial recognition against a known list of faces. If the face is a match, it will announce the name of the person, as well. An example of vocal guidance would be "Elizabeth near top edge, three feet away," as demonstrated in Figure 14-24.

To identify a person's location in the camera feed, the system is running on a fas mobile-optimized object detector. A crop of this face is then passed for further processing to age, emotion, hair style, and other recognition algorithms on the cloud from Microsoft's Cognitive Services. For identifying people in a privacy-friendly way, the app asks the user's friends and family to take three selfies of their faces and generates a feature representation (embedding) of the face that is stored on the device (rather than storing any images). When a face is detected in the camera live feed in the future, an embedding is calculated and compared witl the database of embeddings and associated names. This is based on the concept of one-shot learning, similar to Siamese networks that we saw in Chapter 4. Another benefit of not storing images is that the size of the app doesn't increase drastically even with a large number of faces stored.

t

n *Figure 14-24. Face detection feature on Seeing AI*

Figure 14-24. Face detection feature on Seeing AI

Autonomous Cars

Several self-driving car companies including Waymo, Uber, Tesla, Cruise, NVIDIA, and others use object detection, segmentation, and other deep learning techniques for building their self-driving cars. Advanced Driver Assistance Systems (ADAS) such as pedestrian detection, vehicle type identification, and speed limit sign recognition are important components of a self-driving car.

For the decision-making involved in self-driving cars, NVIDIA uses specialized networks for different tasks. For example, WaitNet is a neural network used for doing fast, high-level detection of traffic lights, intersections, and traffic signs. In addition to being fast, it is also meant to withstand noise and work reliably in tougher conditions such as rain and snow. The bounding boxes detected by WaitNet are then fed to specialized networks for more detailed classification. For instance, if a traffic light is detected in a frame, it is then fed into LightNet—a network used for detecting traffic light shape (circle versus arrow) and state (red, yellow, green). Additionally, if a traffic sign is detected in a frame, it's fed into SignNet, a network to classify among a few hundred types of traffic signs from the US and Europe, including stop signs, yield signs, directional information, speed limit information, and more. Cascading output from faster networks to specialized networks helps improve performance and modularize development of different networks.

Figure 14-25. Detecting traffic lights and signs on a self-driving car using the NVIDIA Drive Platform (image source)

ı

r

,

Summary

In this chapter, we explored the types of computer-vision tasks, including how they relate to one another and how they are different from one another. We took a deeper look at object detection, how the pipeline works, and how it evolved over time. We then collected data, labeled it, trained a model (with and without code), and deployed the model to mobile. We next looked at how object detection is being used in the real world by industry, government, and non-governmental organizations (NGOs). And then we took a sneak peek into the world of object segmentation. Object detection is a very powerful tool whose vast potential is still being discovered every day in numerous areas of our lives. What innovative use of object detection can you think of? Share it with us on Twitter @PracticalDLBook.

Summary

In this chapter, we explored the types of computer-vision tasks, including how they relate to one another and how they are different from one another. We took a deeper look at object detection, how the pipeline works, and how it evolved over time. We then collected data, labeled it, trained a model (with and without code), and deployed the model to mobile. We next looked at how object detection is being used in the real world by industry, government, and non-governmental organizations (NGOs). And then we took a sneak peek into the world of object segmentation. Object detection is a very powerful tool whose vast potential is still being discovered every day in numerous areas of our lives. What innovative use of object detection can you think of? Share it with us on Twitter @PracticalDLBook.

Chapter 15. Becoming a Maker: Exploring Embedded AI at the Edge

Contributed by guest author: Sam Sterckval

You know how to build a great AI application, but you want more. You don't want to be limited to just running AI software on some computer, you want to bring it out in the real physical world. You want to build devices to make things more interactive, to make life easier, to serve humanity, or perhaps just for the fun of it. Maybe you want to build an interactive painting that smiles at you when you look at it. A camera on your door that makes a loud alarm when an unauthorized person attempts to steal delivered packages. Maybe a robotic arm that sorts recyclables and trash. A device in the woods to prevent wildlife poaching, perhaps? Or a drone that can autonomously survey large areas and identify people in distress during floods. Maybe even a wheelchair that could drive on its own. What you need is a smart electronic device, but how would you build it, what would it cost, how powerful would it be? In this chapter, we begin to address those questions.

We look at how to implement AI on an embedded device—a device that you might use in a "maker" project. Makers are people with a DIY spirit who use their creativity to build something new. Often starting as amateur hobbyists, makers are fun-loving problem solvers, roboticists, innovators, and sometimes entrepreneurs.

The aim of this chapter is to spark your ability to select the appropriate device for the task (which means not trying to run a heavy GAN on a tiny CPU or get a quadrillion-core GPU to run a "Not Hotdog" classifier), and set it up for the tests as quickly and easily as possible. We do this by exploring a few of the better-known devices out there, and seeing how we can use them to perform inferencing of our model. And finally, we look at how makers around the world are using AI to build robotic projects.

Let’s take our first step and look at the current landscape of embedded AI devices.

–

1

3

Let’s take our first step and look at the current landscape of embedded AI devices.

Exploring the Landscape of Embedded AI Devices

In this section, we explore a few well-known embedded AI devices, listed in Table 15-1. We talk about their inner workings and the differences between them before we go into testing.

Table 15-1. Device list

Raspberry Pi 4	The most famous single-board computer, as of this writing
Intel Movidius NCS2	A USB accelerator using a 16-core Visual Processing Unit (VPU)
Google Coral USB	A USB accelerator using a custom Google Application-Specific Integrated Circuit (ASIC)
NVIDIA Jetson Nano	A single-board computer using a combination of CPU and 128-core CUDA GPU
PYNQ-Z2	A single-board computer using the combination of CPU and 50k CLB Field-Programmable Gate Array (FPGA)

Instead of saying one is better than the other, we want to learn how to choose a setup for a particular project. We don't typically see a Ferrari during the morning commute. Smaller, more compact cars are more common, and they get the job done just as well, if not better. Similarly, using a powerful $1,000-plus NVIDIA 2080 Ti GPU might be overkill for a battery-powered drone. Following are some questions we should be asking ourselves to understand which of these edge devices would best suit our needs:

1. How big is the device (for instance, compared to a coin)?
2. How much does the device cost? Consider whether you're on a tight budget.
3. How fast is the device? Will it process a single FPS or 100 FPS?
4. How much power (in watts) does the device typically need? For battery-powered projects, this can be essential.

With these questions in mind, let's explore some of the devices in Figure 15-1 one by one.

Figure 15-1. Family photo of Embedded AI devices; starting at the top, going clockwise: PYNQ-Z2, Arduino UNO R3, Intel Movidius NCS2, Raspberry Pi 4, Google Coral USB Accelerator, NVIDIA Jetson Nano, and a €1 coin for reference in the middle

With these questions in mind, let's explore some of the devices in Figure 15-1 one by one.

Figure 15-1. Family photo of Embedded AI devices; starting at the top, going clockwise: PYNQ-Z2, Arduino UNO R3, Intel Movidius NCS2, Raspberry Pi 4, Google Coral USB Accelerator, NVIDIA Jetson Nano, and a €1 coin for reference in the middle

Raspberry Pi

Because we will be talking about embedded devices for makers, let's begin with the one universally synonymous with electronic projects: the Raspberry Pi (Figure 15-2). It's cheap, it's easy to build on, and it has a huge community.

Figure 15-2. Raspberry Pi 4

Size	85.6 mm x 56.5 mm
Price	Starting at $35
Processing unit	ARM Cortex-A72 CPU
Power rating	15W

First of all, what is a Raspberry Pi? It's a "single board computer," which simply means that all calculations can be performed on a single printed circuit board (PCB). The fourth version of the single-board computer houses a Broadcom SoC (system-on-a-chip) containing (most important) an ARMv8-A72 quad-core

CPU, a VideoCore VI 3D unit, and some video decoders and encoders. It comes with a choice of RAM size, up to 4 GB.

This version is a big step up from the Raspberry Pi 3 in terms of performance, but it does lose a bit of efficiency. This is due to the Raspberry Pi 3 having an ARMv8-A53 core, which is the high-efficiency version, whereas the ARMv8-A72 core (used in version 4) is the high-performance version. We can see this reflected in the power supply recommendation, which was 2.5 amps for the Raspberry Pi 3. For the Raspberry Pi 4, this became 3 amps. It is also important to note that the Raspberry Pi 4 has USB 3 ports, where the Raspberry Pi 3 only has USB 2 ports. This will prove to be important information in the future.

On to some machine learning stuff now. However powerful the Raspberry Pi 4 has become, it still mainly consists of four sequential CPU cores (which now support Out of Order [OoO] execution). It has the VideoCore VI, but there is no TensorFlow version available for this architecture as of this writing. Koichi Nakamura from Idein Inc. has built py-videocore, a Python library for accessing the Quad Processing Units (QPUs; the GPU-like cores of the Raspberry Pi's SoC). He has accelerated neural networks with it before, but simple acceleration of TensorFlow isn't possible yet. C++ libraries for accessing these cores are also available. But as you might suspect, these cores are not that powerful, so even when used to accelerate neural networks, they might not yield the desired results For the sake of simplicity, we will not go into these libraries, because digging into the algorithms is beyond the scope of this book. And, and as we will see further on, this might not be necessary at all.

In the end, the Raspberry Pi has proven to be an immensely useful piece of hardware for numerous tasks. It is often used for educational purposes, and by makers. You would be surprised how many industries use the Raspberry Pi in industrial environments (there is a thing called the netPi, which is just a Raspberry Pi with a robust enclosure).

CPU, a VideoCore VI 3D unit, and some video decoders and encoders. It comes with a choice of RAM size, up to 4 GB.

This version is a big step up from the Raspberry Pi 3 in terms of performance, but it does lose a bit of efficiency. This is due to the Raspberry Pi 3 having an ARMv8-A53 core, which is the high-efficiency version, whereas the ARMv8-A72 core (used in version 4) is the high-performance version. We can see this reflected in the power supply recommendation, which was 2.5 amps for the Raspberry Pi 3. For the Raspberry Pi 4, this became 3 amps. It is also important to note that the Raspberry Pi 4 has USB 3 ports, where the Raspberry Pi 3 only has USB 2 ports. This will prove to be important information in the future.

On to some machine learning stuff now. However powerful the Raspberry Pi 4 has become, it still mainly consists of four sequential CPU cores (which now support Out of Order [OoO] execution). It has the VideoCore VI, but there is no TensorFlow version available for this architecture as of this writing. Koichi Nakamura from Idein Inc. has built py-videocore, a Python library for accessing the Quad Processing Units (QPUs; the GPU-like cores of the Raspberry Pi's SoC). He has accelerated neural networks with it before, but simple acceleration of TensorFlow isn't possible yet. C++ libraries for accessing these cores are also available. But as you might suspect, these cores are not that powerful, so even when used to accelerate neural networks, they might not yield the desired results. For the sake of simplicity, we will not go into these libraries, because digging into the algorithms is beyond the scope of this book. And, and as we will see further on, this might not be necessary at all.

In the end, the Raspberry Pi has proven to be an immensely useful piece of hardware for numerous tasks. It is often used for educational purposes, and by makers. You would be surprised how many industries use the Raspberry Pi in industrial environments (there is a thing called the netPi, which is just a Raspberry Pi with a robust enclosure).

Intel Movidius Neural Compute Stick

Let's now dive straight into one of the reasons the Raspberry Pi is the go-to base for a lot of projects: The Intel Movidius Neural Compute Stick 2 (Figure 15-3), the second generation of a USB accelerator created by Intel.

Figure 15-3. Intel Neural Compute Stick 2

Size	72.5 mm x 27 mm
Price	$87.99
Processing unit	Myriad X VPU
Power rating	1W

What it does is pretty simple: you give it a neural network and some data, and it does all the calculations necessary for inference. And it's connected only througl USB, so you can just hook it up to your Raspberry Pi, run your inferencing on the USB accelerator, and free up the CPU of your Raspberry Pi to do all the

other cool stuff you want it to do.

It is based on the Myriad VPU. The first generation had a Myriad 2 VPU, this one has a Myriad X VPU. The VPU contains 16 SHAVE (Streaming Hybrid Architecture Vector Engine) cores, which are kind of like GPU cores but less tailored to graphics-related operations. It features on-chip RAM, which is especially useful when doing neural network inferencing because these networks tend to create a lot of data while they compute, which then can be stored right next to the core, which reduces access time drastically.

other cool stuff you want it to do.

It is based on the Myriad VPU. The first generation had a Myriad 2 VPU, this one has a Myriad X VPU. The VPU contains 16 SHAVE (Streaming Hybrid Architecture Vector Engine) cores, which are kind of like GPU cores but less tailored to graphics-related operations. It features on-chip RAM, which is especially useful when doing neural network inferencing because these networks tend to create a lot of data while they compute, which then can be stored right next to the core, which reduces access time drastically.

Google Coral USB Accelerator

The Google Coral (Figure 15-4), containing the Google Edge TPU, is the second USB accelerator we will discuss. First, a little explanation on why we are looking at two different USB accelerators. The Intel stick, as mentioned, has a number of SHAVE cores, which have multiple available instructions, like GPU cores, and thus act as a processing unit. The Google Edge TPU, on the other hand, is an ASIC (Application Specific Integrated Circuit), which also does some processing, but it serves a single purpose (hence the "Specific" keyword). An ASIC comes with a few properties inherent to the hardware, some of which are really nice, others less so:

Speed

Because all electronic circuits inside the TPU serve a single purpose, there is no overhead in terms of decoding operations. You pump in input data and weights, and it gives you a result, almost instantly.

Efficiency

All ASICs serve a single purpose, so no extra energy is required. The performance/watt figure for ASICs is usually the highest in the business.

Flexibility

An ASIC can do only what it was designed for; in this case, that would be accelerating TensorFlow Lite neural networks. You will need to stick to the Google Edge TPU compiler and 8-bit *.tflite* models.

Complexity

Google is, in essence, a software company and it knows how to make things easy to use. That is exactly what it has done here, as well. The Google Coral is incredibly easy to get started with.

So how does the Google Edge TPU work? Information on the Edge TPU itself has not been shared, but information about the Cloud TPU has, so the assumption is that the Edge TPU works in broadly the same way as the Cloud TPU. It has dedicated hardware to do the multiply-add, activation, and pooling operations. All the transistors on the chip have been connected in such a way

that it can take weights and input data, and it will calculate the output in a highly parallel fashion. The single biggest part of the chip (apart from the on-chip memory, that is) is a part that does exactly what it sounds like: the “Matrix Multiply Unit.” It uses a rather clever, though not so new, principle called *systolic execution*. This execution principle helps lower memory bandwidth by storing intermediate results in the processing elements rather than in memory.

Figure 15-4. Google Coral USB Accelerator

Size	65 mm x 30 mm
Price	$74.99
Processing unit	Google Edge TPU
Power rating	2.5W

NOTE

What’s the main difference between the two USB accelerators we’ve discussed? The Google Coral is more powerful, but a (little) less flexible than the Intel Movidius NCS2. That being said, the Coral is far

easier to set up and work with, certainly after you have the trained and converted model.

easier to set up and work with, certainly after you have the trained and converted model.

NVIDIA Jetson Nano

On to a different kind of hardware: the NVIDIA Jetson Nano (Figure 15-5), a single board AI computer. Kind of like the Raspberry Pi, but with a 128-core CUDA-enabled Maxwell GPU. The addition of the GPU makes it kind of simila to a Raspberry Pi with an Intel NCS2, but where the NCS2 has 16 cores, this Jetson has 128. What more does it contain? A quad-core A57 ARMv8 CPU, which is the predecessor of the ARMv8 A72 in the Raspberry Pi 4 (it is also a few months older than the Raspberry Pi 4), 4 GB of Low-Power Double Data Rate 4 (LPDDR4) RAM memory, which is, quite conveniently, shared between the CPU and GPU, which allows you to process data on the GPU, without copying it.

Figure 15-5. NVIDIA Jetson Nano

Size	100 mm x 79 mm
Price	$99.00

Processing unit	ARM A57 + 128 core Maxwell GPU
Power rating	10W (Can spike higher under high load)

The thing about this single board computer is that, as said before, these GPU cores are CUDA enabled, so yes, they can run TensorFlow-GPU or any-other-framework, which will make a big difference compared to a Raspberry Pi, especially when you also plan to train networks. Also, it's dirt cheap for a GPU. At this point in time, the Jetson Nano is yours for $99, and this includes a rather high-performance ARM CPU and a 128-core GPU. In comparison, the Raspberry Pi 4 with 4 GB of memory is around $55, the Coral USB accelerator is around $75, and the Movidius NCS2 is about $90, as well. The latter two are not standalone, and will at least need an additional Raspberry Pi to actually do something, and the Pi has no GPU that can easily accelerate deep learning applications.

One more note about the Jetson Nano: it can accelerate default TensorFlow 32-bit floating-point operations, but it will get much more efficient when 16-bit floating-point operations are used, and even more efficient if you use its own TensorRT framework. Luckily, the company has a nice little open source library called TensorFlow-TensorRT (TF-TRT) that will accelerate the available operations with TensorRT automatically while allowing TensorFlow to do the rest. This library offers grand speedups of around four times compared to TensorFlow-GPU. With all this in mind, this makes the Jetson Nano easily the most flexible device.

FROM THE CREATORS' DESK

By John Welsh and Chitoku Yato, NVIDIA Jetson Nano Team

It's exciting to be working on the NVIDIA Jetson platform, which provides the software and hardware capabilities to drive AI-powered robots, drones, Intelligent Video Analytics (IVA) applications, and other autonomous machines that think for themselves. Jetson Nano brings the power of modern AI to a small, easy-to-use platform.

A few years ago, we had a group of high school interns who were developing summer projects using deep learning at the edge. The projects

they came up with were creative, technically impressive, and fun. We were very excited to come out with Jetson Nano, because we felt that these kinds of projects were exactly what Jetson Nano would enable for a larger audience.

It’s really rewarding to be part of the talented team here at NVIDIA that’s bringing the tools to inspire these communities to get up and running fast with AI. We’re continually amazed by some of the cool projects that makers, developers, and learners are building on Jetson Nano—from DIY robocars to home automation—and it’s just the beginning. Our goal is to make AI more accessible to all.

they came up with were creative, technically impressive, and fun. We were very excited to come out with Jetson Nano, because we felt that these kinds of projects were exactly what Jetson Nano would enable for a larger audience.

It’s really rewarding to be part of the talented team here at NVIDIA that’s bringing the tools to inspire these communities to get up and running fast with AI. We’re continually amazed by some of the cool projects that makers, developers, and learners are building on Jetson Nano—from DIY robocars to home automation—and it’s just the beginning. Our goal is to make AI more accessible to all.

FPGA + PYNQ

Now is the time to fasten your seatbelts, because we are about to take a deep dive into the dark world of electronics. The PYNQ platform (Figure 15-6), based on the Xilinx Zynq family of chips is, for the most part, a totally different side of the electronics world compared to the other devices discussed in this topic. If you do a bit of research, you'll find out it has a dual-core ARM-A9 CPU, at a whopping 667 MHz. The first thing you'll think is "Are you serious? That is ridiculous compared to the 1.5 GHz quad-core A72 from the Raspberry Pi?!" And you'd be right, the CPU in this thing is, for the most part, absolutely worthless. But it has something else on board—an FPGA.

Figure 15-6. Xilinx PYNQ-Z2

Size	140 mm x 87 mm
Price	$119.00
Processing unit	Xilinx Zynq-7020

Power rating	13.8W

FPGAs

To understand what an FPGA is, we need to first look at some familiar concepts. Let's begin with a CPU: a device that knows a list of operations known as the Instruction Set Architecture (ISA). This is a set that defines everything the CPU can do, and usually contains operations such as "Load Word" (a word is typically a number of bits equal to the datapath size of the CPU, usually 32-bit o 64-bit), which will load some value into an internal register of the CPU, and "Add," which can add up two of the internal registers, and store the result in a third register, for example.

The reason the CPU can do this is because it contains a whole bunch of transistors (look at these as electrical switches if you like) that are hardwired in such a way that the CPU automatically translates the operations and does whatever that operation was intended to do. A software program is just a really long list of these operations, in a precisely thought-out order. A single-core CPU will take in operation per operation and carry them out, at pretty impressive speeds.

Let's look at parallelism. Neural networks, as you know, consist of mostly convolutions or linear nodes, which can all be translated to matrix multiplications. If we look at the mathematics behind these operations, we can spot that each output point of a layer can be calculated independently from the other output points, as demonstrated in Figure 15-7.

Figure 15-7. Matrix multiplication

We can now see that all of these operations could actually be done in a parallel fashion. Our single-core CPU won't be able to do so, because it can do only one operation at a time, but that's where multicore CPUs come in. A quad-core CPU can do four operations at a time, which means a theoretical speedup of four times. The SHAVE cores in the Intel Movidius NCS2 and the CUDA cores in the Jetson Nano might not be as complex as a CPU core, but they are good enough for these multiplications and additions, and instead of having four of these, the NCS2 has 16, and the Jetson Nano has 128. A bigger GPU like a RTX 2080 Ti even has 4,352 CUDA cores. It's easy to see now why GPUs are better at performing deep learning tasks compared to CPUs.

Let's get back to FPGAs. Whereas CPUs and GPUs are huge collections of transistors hardwired to carry out a set of instructions, you can think of an FPGA as that same huge collection of transistors, but not wired. You can choose how they are wired, and rewire them whenever you want, making them reconfigurable. You can wire them to be a CPU. You can also find schematics and projects for which people have wired them to be a GPU. But most interesting here is that they can even be wired to the exact same architecture as your deep learning neural network, which would actually make them a physical implementation of your network. The word "wired" is used intentionally here. Often, you'll find people talking about this configuration with the word

“program,” but this can be confusing. What you’re doing is reconfiguring the actual hardware, unlike a CPU or GPU for which you download a program that the hardware can run.

PYNQ platform

We usually call the “wiring” file for an FPGA a *bitstream* or a *bitmap*, because what you flash onto the chip is basically just a map of the connections that should be made. As you can imagine, making these bitstreams is quite a lot more complicated than running a Python script. That’s where PYNQ comes in. Its tagline is “Python productivity for Zynq.” The company installs the PYNQ image that automatically runs a Jupyter Notebook on the ARM inside the chip, and it comes with a few basic bitstreams; however, more bitstreams will likely become available in the future. Within the PYNQ world, these bitstreams are called *overlays*.

If you go looking for examples, you’ll quickly find an example called “BNN-PYNQ,” which has a simple VGG-style, six-layer CNN that can run at 3,000-ish FPS on the PYNQ-Z1 and Z2, and close to 10,000 FPS on the ZCU104, which has the Ultrascale+ version of the Zynq chip onboard. These numbers look pretty insane, but take into account that they run on 32x32 pixel images, rather than the usual 224x224 pixel images and that this network is “binarized,” which means it has weights and activations of one bit, instead of the 32-bits of TensorFlow. To have a better comparison of performance, I tried to recreate a similar network in Keras.

I built the FP32 model and trained it on the CIFAR-10 dataset. The CIFAR-10 dataset is easily available with Keras using `keras.datasets.cifar10`. The FP32 model reached a roughly 17% error rate, which is, surprisingly, only 2% better than the binary model. Inference speeds are around 132 FPS on an Intel i9 eight-core CPU. There is, to my knowledge, no easy way of using a binary network efficiently and easily on a CPU—you’ll need to dig into some specific Python packages or some C code to get the most out of the hardware. You could potentially achieve a speedup of three to five times. This would, however, still be far less than a low-end FPGA, and a CPU will usually draw more power in doing so. Of course, everything has a flip side, and for FPGAs that has to be the complexity of the design. Open source frameworks exist, namely the FINN

framework, backed by Xilinx. Other manufacturers offer additional frameworks, but none of them come close to how easy to use software packages and frameworks like TensorFlow are. Designing a neural network for an FPGA would also involve a lot of electronics knowledge, and thus is beyond the scope of this book.

framework, backed by Xilinx. Other manufacturers offer additional frameworks, but none of them come close to how easy to use software packages and frameworks like TensorFlow are. Designing a neural network for an FPGA would also involve a lot of electronics knowledge, and thus is beyond the scope of this book.

Arduino

Arduino (Figure 15-8) can make your life a lot easier when trying to interact with the real world through sensors and actuators.

AnMicroController Units (MCUs) Arduino is a microcontroller development board built around the 8-bit Advanced Virtual RISC (AVR) ATMega328p microcontroller running at 16 MHz (so yeah, let's not try to run a decent neural network on that). A microcontroller is something everyone has come into contac with—you can find microcontrollers in almost everything that is slightly electronic, ranging from every screen/TV, to your desktop keyboard, heck bicycle lights may even contain a microcontroller to enable the flashing mode.

These boards come in a variety of shapes, sizes, and performances, and although the MCU in the Arduino UNO is pretty low performance, more potent MCUs ar available. The ARM Cortex-M series is probably the most well known. Recently TensorFlow and uTensor (an extremely lightweight machine learning inference framework aimed at MCUs) joined forces to enable the possibility to run TensorFlow models on these MCUs.

FROM THE CREATOR'S DESK

By Pete Warden, technical lead for Mobile and Embedded TensorFlow, author of *TinyML*

When I joined Google in 2014, I was amazed to find that the "Ok Google" team was using models that were only 13 KB in size, and ran on tiny DSPs (Digital Signal Processors), to spot wake words. They were very experienced and pragmatic engineers, so I knew they weren't doing this to be fashionable; they'd experimented and this was the best solution they could find. At that time, I was used to models being megabytes in size, so finding out that tiny models could be useful was a revelation, and I couldn't help wondering what other problems they could be useful for. Since then, I've been keen to open up the technology they pioneered to a wider audience so we can discover what some of these applications might be. This is why we released our first version of TensorFlow Lite for Microcontrollers in early 2019, and it has been amazing to hear from all of the creative engineers

building innovative new features, and even entirely new products, using these new capabilities.

Figure 15-8. Arduino UNO R3

Size	68.6 mm x 53.3 mm
Price	Less than $10.00
Processing unit	AVR ATMega328p
Power rating	0.3W

Although the Arduino UNO is not really useful for performing lots of calculations, it has many other benefits that make it a mandatory part of the workbench of a maker. It’s cheap, it’s simple, it’s reliable, and it has a huge community surrounding it. Most sensors and actuators have some kind of library and shield for Arduino, which makes it a really easy platform to interact with. The AVR architecture is a kind of dated 8-bit modified Harvard architecture, but it’s really easy to learn, especially with the Arduino framework on top of it. The

huge community around Arduino obviously brings a lot of tutorials with it, so just look up any Arduino tutorial for your sensor of choice, and you'll surely find what you need for your project.

TIP

Using Python's serial library, you can easily interface with an Arduino using a Universal Asynchronous Receiver/Transmitter (UART) through a USB cable to send commands or data back and forth.

huge community around Arduino obviously brings a lot of tutorials with it, so just look up any Arduino tutorial for your sensor of choice, and you'll surely find what you need for your project.

TIP

Using Python's serial library, you can easily interface with an Arduino using a Universal Asynchronous Receiver/Transmitter (UART) through a USB cable to send commands or data back and forth.

A Qualitative Comparison of Embedded AI Devices

Table 15-2 summarizes the platforms we've talked about, and Figure 15-9 plots performance versus complexity for each device.

Table 15-2. Pros and cons of tested platforms

Embedded device	Pros	Cons
Raspberry Pi 4	• Easy • Large community • Readily available • Cheap	• Lacks computing power
Intel Movidius NCS2	• Easy optimizations • Supports most platforms	• Rather expensive for the speedup • Not so powerful
Google Coral USB	• Easy getting started • Huge speedup • Price versus speedup	• Supports only *.tflite* • Needs eight-bit quantization
NVIDIA Jetson Nano	• All-in-one • Training is possible • Cheap • Really easy to achieve decent performance • CUDA GPU	• Performance might still be lacking • Advanced optimizations can be complex
PYNQ-Z2	• Potentially huge power efficiency • Actual hardware design	• Massively complex • Long design flow

- Potentially huge performance
- Even more versatile than the Jetson Nano

Figure 15-9. Performance versus complexity-to-use (the size of the circle represents price)

In this section, we attempt to test some platforms by performing 250 classifications of the same image using MobileNetV2, with the top platform trained on the ImageNet dataset. We use the same image every time because this will allow us to eliminate part of the data bottlenecks that can occur in systems that don't have enough RAM to store all weights and lots of different images.

Ok, let's find an image first. Who doesn't like cats? So, let's have an image of a beautiful cat (Figure 15-10) that is exactly 224 x 224 pixels, that way we don't need to scale the image.

Figure 15-10. The image of the cat we will be using for our experiments

It's always nice to have a benchmark, so let's run the model on our PC first. Chapter 2 explains how to do this, so let's go ahead and write a script that will invoke the prediction 250 times, and measure how long this takes. You can find the full script on the book's GitHub website (see *http://PracticalDeepLearning.ai*) at *code/chapter-15*:

```
$ python3 benchmark.py
Using TensorFlow backend.
tf version : 1.14.0
keras version : 2.2.4
input tensor shape : (1, 224, 224, 3)
warmup prediction
[('n02124075', 'Egyptian_cat', 0.6629321)]
starting now...
Time[s] : 16.704
FPS     : 14.968
Model saved.
```

Now that we've established our benchmark, it's time to begin looking at how to run this model on the embedded devices and see whether and how we can take the performance to a useful level.

Figure 15-10. The image of the cat we will be using for our experiments

It's always nice to have a benchmark, so let's run the model on our PC first. Chapter 2 explains how to do this, so let's go ahead and write a script that will invoke the prediction 250 times, and measure how long this takes. You can find the full script on the book's GitHub website (see *http://PracticalDeepLearning.ai*) at *code/chapter-15*:

```
$ python3 benchmark.py
Using TensorFlow backend.
tf version : 1.14.0
keras version : 2.2.4
input tensor shape : (1, 224, 224, 3)
warmup prediction
[('n02124075', 'Egyptian_cat', 0.6629321)]
starting now...
Time[s] : 16.704
FPS     : 14.968
Model saved.
```

Now that we've established our benchmark, it's time to begin looking at how to run this model on the embedded devices and see whether and how we can take the performance to a useful level.

Hands-On with the Raspberry Pi

We begin with the most well-known and most basic of the previously discussed devices: the Raspberry Pi (RPi for short).

We start by installing Raspbian, a Linux variant made especially for the Raspberry Pi. For the sake of brevity, let's assume that we have a Raspberry Pi with everything installed, updated, connected, and ready. We can go straight to installing TensorFlow on the Pi, which should work using the `pip` installer:

```
$ pip3 install tensorflow
```

NOTE

Due to the recent switch to Raspbian Buster, we encountered some problems, which were resolved by using the following `piwheel`:

```
$ pip3 install
https://www.piwheels.org/simple/tensorflow/tensorflow
-1.13.1-cp37-none-linux_armv7l.whl
```

Installing Keras is straightforward with `pip`, but don't forget to install `libhdf5-dev`, which you will need if you want to load weights into a neural network:

```
$ pip3 install keras
$ apt install libhdf5-dev
```

Because installing OpenCV, called in code as cv2, on the RPi can be a burden (especially when a recent switch of OS has happened), we can load the image using the PIL instead of OpenCV. This means replacing the import of cv2 with an import for PIL, and changing the code to load the image to use this library:

```
#input_image = cv2.imread(input_image_path)
#input_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB)
input_image = PIL.Image.open(input_image_path)
input_image = np.asarray(input_image)
```

TIP

Unlike OpenCV, PIL loads images in the RGB format, so a conversion from BGR to RGB is no longer needed.

And that's it! The exact same script should run on your Raspberry Pi now:

```
$ python3 benchmark.py
Using TensorFlow backend.
tf version : 1.14.0
keras version : 2.2.4
input tensor shape : (1, 224, 224, 3)
warmup prediction
[('n02124075', 'Egyptian_cat', 0.6629321)]
starting now...
Time[s] : 91.041
FPS     : 2.747
```

As you can see, it runs a lot slower, dropping to less than three FPS. Time to think about how we can speed this up.

The first thing we could do is have a look at TensorFlow Lite. TensorFlow has, as discussed in Chapter 13, a converter built in, which lets us easily convert any model to a TensorFlow Lite (*.tflite*) model.

We proceed by writing a small script, very similar to the original benchmark script, which will just set up everything to use the TensorFlow Lite model and run the 250 predictions. This script is, of course, also available on the book's GitHub webiste (see *http://PracticalDeepLearning.ai*).

Let's go ahead and run this so that we can evaluate our effort to speed things up.

```
$ python3 benchmark_tflite.py
Using TensorFlow backend.
input tensor shape : (1, 224, 224, 3)
conversion to tflite is done
INFO: Initialized TensorFlow Lite runtime.
[[('n02124075', 'Egyptian_cat', 0.68769807)]]
starting now (tflite)...
Time[ms] : 32.152
FPS      : 7.775
```

NOTE

The Google Coral website has 8-bit quantized CPU models available, which you can run using TensorFlow Lite.

For the Raspberry Pi 4, we observed an increase in speed of three times (see Table 15-3). Not bad for a quick try, but we still only achieved 7.8 FPS, which is kind of nice and perfectly fine for a lot of applications. But what if our original plan was to do something on live video, for example? Then this is not enough. Quantizing the model can achieve even more performance, but this will be only a marginal increase given that the CPU in the Raspberry Pi has not been optimized for 8-bit integer operations.

Table 15-3. Raspberry Pi benchmarks

Setup	FPS
Raspberry Pi 4 (tflite, 8-bit)	8.6
Raspberry Pi 4 (tflite)	7.8
Raspberry Pi 3B+ (tflite, 8-bit)	4.2
Raspberry Pi 4	2.7
Raspberry Pi 3B+	2.1

So how can we speed things up even more so that they become useful for applications like autonomous racecars or automatic drones? We touched on it before: the USB accelerators.

> **NOTE**
>
> The Google Coral website has 8-bit quantized CPU models available, which you can run using TensorFlow Lite.

For the Raspberry Pi 4, we observed an increase in speed of three times (see Table 15-3). Not bad for a quick try, but we still only achieved 7.8 FPS, which is kind of nice and perfectly fine for a lot of applications. But what if our original plan was to do something on live video, for example? Then this is not enough. Quantizing the model can achieve even more performance, but this will be only a marginal increase given that the CPU in the Raspberry Pi has not been optimized for 8-bit integer operations.

Table 15-3. Raspberry Pi benchmarks

Setup	FPS
Raspberry Pi 4 (tflite, 8-bit)	8.6
Raspberry Pi 4 (tflite)	7.8
Raspberry Pi 3B+ (tflite, 8-bit)	4.2
Raspberry Pi 4	2.7
Raspberry Pi 3B+	2.1

So how can we speed things up even more so that they become useful for applications like autonomous racecars or automatic drones? We touched on it before: the USB accelerators.

Speeding Up with the Google Coral USB Accelerator

That's right, these accelerators come in quite handy right now. All of our hardware setup can stay exactly the same, we just need to add a little bit here and there. So let's find out: how do we get the Google Coral USB Accelerator to run on the Raspberry Pi, and will it speed things up?

First things first. Assuming that we have a USB accelerator on our hands, the first thing we should always do is look for a "Getting Started" guide from the vendor. Check out *https://coral.withgoogle.com/*, head to *Docs>USB Accelerator>Get Started*, and you're rolling in mere minutes.

As of this writing, the Coral USB Accelerator is not yet fully compatible with the Raspberry Pi 4, but after fiddling around with the *install.sh* script, it is pretty easy to get it running. You need to simply add a part to the install script so that it recognizes the Pi 4, as demonstrated in Figure 15-11.

```
    LIBEDGETPU_SUFFIX=arm32
    HOST_GNU_TYPE=arm-linux-gnueabihf
  elif [[ "${MODEL}" == "Raspberry Pi 3 Model B Plus Rev"* ]]; then
    info "Recognized as Raspberry Pi 3 B+."
    LIBEDGETPU_SUFFIX=arm32
    HOST_GNU_TYPE=arm-linux-gnueabihf
  elif [[ "${MODEL}" == "Raspberry Pi 4 Model B Rev"* ]]; then
    info "Recognized as Raspberry Pi 4."
    LIBEDGETPU_SUFFIX=arm32
    HOST_GNU_TYPE=arm-linux-gnueabihf
  fi
elif [[ "${CPU_ARCH}" == "aarch64" ]]; then
  info "Recognized as generic ARM64 platform."
  LIBEDGETPU_SUFFIX=arm64
  HOST_GNU_TYPE=aarch64-linux-gnu
```

Figure 15-11. Google Coral install script changes

Now the *install.sh* script will run correctly. Then, we need to rename a *.so* file, so that it will also work with Python 3.7. To do this, use the Unix copy command:

```
$ sudo cp /usr/local/lib/python3.7/dist-
packages/edgetpu/swig/_edgetpu_cpp_wrapper.cpython-
```

```
35m-arm-linux-gnueabihf.so
/usr/local/lib/python3.7/dist-
packages/edgetpu/swig/_edgetpu_cpp_wrapper.cpython-
37m-arm-linux-gnueabihf.so
```

After you do this, everything should work correctly, and the demonstrations from Google should run.

Now that we have the Coral running, let's try to benchmark the same MobileNetV2 model again. But this time, it must be the quantized version of it, because Google's Edge TPU supports only 8-bit integer operations. Google supplied us with the quantized and converted MobileNetV2 model, ready to use with the Edge TPU. We can download it from the Coral website under *Resources > See pre-compiled models > MobileNetV2(ImageNet) > Edge TPU Model*.

> **NOTE**
>
> If you want to create, train, quantize, and convert your own model for the Edge TPU, this is a bit more work and cannot be done using Keras. You can find information on how to do this on the Google Coral website.

We have a working USB accelerator and a model to run on it. Now, let's make a new script to test its performance. The file we are about to make is again very similar to the previous one and takes a lot straight from the examples Google supplied with the Coral. And, as always, this file is available for download at GitHub website (see *http://PracticalDeepLearning.ai*):

```
$ python3 benchmark_edgetpu.py
INFO: Initialized TensorFlow Lite runtime.
warmup prediction
Egyptian cat
0.59375
starting now (Edge TPU)...
Time[s] : 1.042
FPS     : 240.380
```

Yes, that is correct. It did all 250 classifications already! For the Raspberry Pi 4 with USB3, it took only 1.04 seconds, which translates to 240.38 FPS! Now that's a speedup. As you can expect with these speeds, live video would be no

problem at all.

Lots of precompiled models are available, for all kinds of different purposes, so check them out. You might be able to find a model that suits your needs so that you don't need to go through the flow (or struggle) to create, train, quantize, and convert your own.

The Raspberry Pi 3 has no USB3 ports, only USB2. We can clearly see in Table 15-4 that this creates a data bottleneck for the Coral.

Table 15-4. Google Coral benchmarks

Setup	FPS
i7-7700K + Coral (tflite, 8-bit)	352.1
Raspberry Pi 4 + Coral (tflite, 8-bit)	240.4
Jetson Nano + Coral (tflite, 8-bit)	223.2
RPi3 + Coral (tflite, 8-bit)	75.5

problem at all.

Lots of precompiled models are available, for all kinds of different purposes, so check them out. You might be able to find a model that suits your needs so that you don't need to go through the flow (or struggle) to create, train, quantize, and convert your own.

The Raspberry Pi 3 has no USB3 ports, only USB2. We can clearly see in Table 15-4 that this creates a data bottleneck for the Coral.

Table 15-4. Google Coral benchmarks

Setup	FPS
i7-7700K + Coral (tflite, 8-bit)	352.1
Raspberry Pi 4 + Coral (tflite, 8-bit)	240.4
Jetson Nano + Coral (tflite, 8-bit)	223.2
RPi3 + Coral (tflite, 8-bit)	75.5

Port to NVIDIA Jetson Nano

That USB accelerator is nice, but what if your project needs a real niche model, trained on some crazy, self-made, dataset? As we said earlier, creating a new model for the Coral Edge TPU is quite a bit more work than just creating a Keras model. So, is there an easy way to have a custom model running on the edge with some decent performance? NVIDIA to the rescue! With its Jetson Nano, the company has created a replacement for the Raspberry Pi that has a CUDA enabled, and rather efficient, GPU on board, which lets you accelerate not only TensorFlow, but anything you like.

Again, we must begin by installing all needed packages. First, you want to download a copy of NVIDIA's JetPack, which is its version of a Debian-like OS for the Jetson platform. Then, to install TensorFlow and the needed packages, NVIDIA walks us through how to do that here.

NOTE

Note there is a known issue with `pip3`: "`cannot import name 'main'`".

Here's the solution:

```
$ sudo apt install nano
$ sudo nano /usr/bin/pip3


Replace : main => __main__
Replace : main() => __main__._main()
```

TIP

Updating all `pip` packages can take a long time; if you want to ensure that your Jetson did not freeze, you might want to first install `htop`:

```
$ sudo apt install htop
$ htop
```

This lets you monitor CPU utilization. As long as this is working, your Jetson is, too.

NOTE

The Jetson Nano tends to run rather hot when compiling, so we recommend using some kind of fan. The development board has a connector, but keep in mind this will feed 5V to the fan, whereas most fans are rated for 12V.

Some 12V fans might do the trick out of the box, some might need a little push to get started. You will need a 40 mm x 40 mm fan. 5V versions are also available.

Unfortunately, the NVIDIA walkthrough is not as easy as the Google one. Chances are you will go through a bit of a struggle with the occasional table-flip moment.

But hang in there; you'll get there eventually.

TIP

Installing Keras on the Jetson requires scipy, which requires `libatlas-base-dev` and `gfortran`, so start by installing the latter, and move to the front:

```
$ sudo apt install libatlas-base-dev gfortran
$ sudo pip3 install scipy
$ sudo pip3 install keras
```

After everything is done, we can choose to run the *benchmark.py* file directly, which will be, because of the GPU, a lot faster than it was on the Raspberry Pi:

```
$ python3 benchmark.py
Using TensorFlow backend.
tf version : 1.14.0
keras version : 2.2.4
input tensor shape : (1, 224, 224, 3)
warmup prediction
[('n02124075', 'Egyptian_cat', 0.6629321)]
starting now...
Time[s] : 20.520
```

```
FPS     : 12.177
```

This immediately shows the power of the Jetson Nano: it runs almost exactly like any other PC with a GPU. However, 12 FPS is still not huge, so let's look at how to optimize this.

TIP

You also can attach the Google Coral to your Jetson Nano, which opens up the possibility to run one model on the Coral, and another one on the GPU simultaneously.

The GPU of the Jetson is actually built specifically for 16-bit-floating point operations, so inherently, this will be the best trade-off between precision and performance. As mentioned earlier, NVIDIA has a package called TF-TRT, which facilitates optimization and conversion. However, it is still a bit more complex than the Coral example (keep in mind that Google supplied us with the precompiled model file for the Coral). You'll need to freeze the Keras model and then create a TF-TRT inference graph. Doing this can take quite a while if you need to find everything laying around on GitHub, so it's bundled on this book's GitHub website (see *http://PracticalDeepLearning.ai*).

You can use the *tftrt_helper.py* file to optimize your own models for inferencing on the Jetson Nano. All it really does is freeze the Keras model, removes the training nodes, and then use NVIDIA's TF-TRT Python package (included in TensorFlow Contrib) to optimize the model.

```
$ python3 benchmark_jetson.py
FrozenGraph build.
TF-TRT model ready to rumble!
input tensor shape : (1, 224, 224, 3)
[[('n02124075', 'Egyptian_cat', 0.66293204)]]
starting now (Jetson Nano)...
Time[s] : 5.124
FPS     : 48.834
```

48 FPS, that's a speedup of four times with just a few lines of code. Exciting, isn't it? You can achieve similar speedups on your own custom models, tailored to the specific needs of your project; these performance benchmarks can be

found in Table 15-5.

Table 15-5. NVIDIA Jetson Nano benchmarks

Setup	FPS
Jetson Nano + Coral (tflite, 8-bit)	223.2
Jetson Nano (TF-TRT, 16-bit)	48.8
Jetson Nano 128CUDA	12.2
Jetson Nano (tflite, 8-bit)	11.3

l

found in Table 15-5.

Table 15-5. NVIDIA Jetson Nano benchmarks

Setup	FPS
Jetson Nano + Coral (tflite, 8-bit)	223.2
Jetson Nano (TF-TRT, 16-bit)	48.8
Jetson Nano 128CUDA	12.2
Jetson Nano (tflite, 8-bit)	11.3

Comparing the Performance of Edge Devices

Table 15-6 shows a quantitative comparison of several edge devices running the MobileNetV2 model.

Table 15-6. Full benchmarking results

Setup	FPS
i7-7700K + Coral (tflite, 8-bit)	352.1
i7-7700K + GTX1080 2560CUDA	304.9
Raspberry Pi 4 + Coral	240.4
Jetson Nano + Coral (tflite, 8-bit)	223.2
RPi3 + Coral (tflite, 8-bit)	75.5
Jetson Nano (TF-TRT, 16-bit)	48.8
i7-7700K (tflite, 8-bit)	32.4
i9-9880HQ (2019 MacBook Pro)	15.0
Jetson Nano 128CUDA	12.2
Jetson Nano (tflite, 8-bit)	11.3
i7-4870HQ (2014 MacBook Pro)	11.1
Jetson Nano (tflite, 8-bit)	10.9
Raspberry Pi 4 (tflite, 8-bit)	8.6
Raspberry Pi 4 (tflite)	7.8
RPi3B+ (tflite, 8-bit)	4.2
Raspberry Pi 4	2.7
RPi3B+	2.1

We can summarize the takeaways from these experiments as follows:

1. Good optimizations make a big difference. The world of computing optimization is still growing rapidly, and we will see big things coming up in the next few years.

2. More is always better when talking about computing units for AI.

3. ASIC > FPGA > GPU > CPU in terms of performance/watt.
4. TensorFlow Lite is awesome, especially for small CPUs. Keep in mind that it is specifically aimed at small CPUs, and using it for x64 machines will not be that interesting.

3. ASIC > FPGA > GPU > CPU in terms of performance/watt.
4. TensorFlow Lite is awesome, especially for small CPUs. Keep in mind that it is specifically aimed at small CPUs, and using it for x64 machines will not be that interesting.

Case Studies

This chapter is about makers. And it would be incomplete without showcasing what they make. Let’s look at a few things that have been created by running AI on the edge.

Case Studies

This chapter is about makers. And it would be incomplete without showcasing what they make. Let’s look at a few things that have been created by running AI on the edge.

JetBot

NVIDIA has been at the forefront of the deep learning revolution, enabling researchers and developers with powerful hardware and software. And in 2019, they took the next step to enable makers, too, by releasing the Jetson Nano. We already know the power of this hardware, so now it's time to build something with it—like a DIY miniature car. Luckily, NVIDIA has us covered with JetBot (Figure 15-12), the open source robot that can be controlled by an onboard Jetson Nano.

The JetBot wiki features beginner-friendly step-by-step instructions on building it. Here's a high-level look:

1. Buy the parts specified in the Bill of Materials, like the motor, caster, camera, and WiFi antenna. There are about 30 parts, costing around $150, on top of the $99 Jetson Nano.

2. It's time to channel our inner MacGyver, get a set of pliers and a cross-tip screwdriver, and assemble the parts. The end result should be something resembling Figure 15-12.

3. Next, we'd set up the software. This involves flashing the JetBot image onto an SD card, booting the Jetson Nano, connecting it to WiFi, and finally connecting to our JetBot from a web browser.

4. Finally, we run through the example notebooks provided. The provided notebooks enable us, through little code, to not only control the bot from a web browser, but also use its camera to collect training data from streaming video, train a deep learning model right on the device (the Jetson Nano is a mini-GPU after all), and use it for tasks such as avoiding obstacles and following objects such as a person or a ball.

Figure 15-12. NVIDIA JetBot

Makers are already using this base framework to extend the JetBot's capabilities such as attaching lidar sensors to it for understanding the environment better, bringing JetBot to different physical forms like tanks, Roomba vacuum cleaners (JetRoomba), object-following spiders (JetSpider), and more. Because all roads eventually lead to racing cars, the Jetson Nano team eventually released a simila open source recipe book for a racing car called JetRacer. It's based on a faster chassis, higher camera frame rate, and optimizations for inference (with TensorRT) to handle the speed. The end result: JetRacer is already being raced a events like the DIY Robocars meetups.

The slowest step here is…waiting for the hardware to arrive.

FROM THE CREATORS' DESK

By John Welsh and Chitoku Yato, NVIDIA Jetson Nano Team

We wanted to develop a reference platform to show developers what's

possible with this tiny piece of hardware—the Jetson Nano. While dreaming up the project that would eventually turn into JetBot, we realized its platform requirements matched what we envisioned many makers would need for their next-level AI projects.

Specifically, JetBot required a platform capable of real-time image processing that worked with an affordable camera and would be easy to interface with maker hardware. As the engineering development of Jetson Nano matured, these boxes were checked. With Jetson Nano as the brain powering JetBot, we felt it would be an excellent platform for makers interested in getting started with AI for a wide range of projects.

We focused on making JetBot easy to use and educational to inspire developers to start creating with Jetson Nano. Although JetBot is shown in NVIDIA demonstrations across the globe, its vision is much more far-reaching. It’s meant to be an example of what’s possible when makers combine our technology with their creativity and ingenuity. Since JetBot’s release, there have been many different spin-offs, and we hope it’s a great guide to learn, explore, and, most important, have fun with AI.

’

r

t

possible with this tiny piece of hardware—the Jetson Nano. While dreaming up the project that would eventually turn into JetBot, we realized its platform requirements matched what we envisioned many makers would need for their next-level AI projects.

Specifically, JetBot required a platform capable of real-time image processing that worked with an affordable camera and would be easy to interface with maker hardware. As the engineering development of Jetson Nano matured, these boxes were checked. With Jetson Nano as the brain powering JetBot, we felt it would be an excellent platform for makers interested in getting started with AI for a wide range of projects.

We focused on making JetBot easy to use and educational to inspire developers to start creating with Jetson Nano. Although JetBot is shown in NVIDIA demonstrations across the globe, its vision is much more far-reaching. It’s meant to be an example of what’s possible when makers combine our technology with their creativity and ingenuity. Since JetBot’s release, there have been many different spin-offs, and we hope it’s a great guide to learn, explore, and, most important, have fun with AI.

Squatting for Metro Tickets

When good health can't motivate people to exercise, what else would? The answer turns out to be…free train tickets! To raise public awareness for its growing obesity crisis, Mexico City installed ticket machines with cameras at subway stations that give free tickets, but only if you exercise. Ten squats get a free ticket (Figure 15-13). Moscow, too, implemented a similar system at its metro stations, but apparently, officials there had a much higher fitness standard at 30 squats per person.

Inspired by these, we could foresee how to build our own "squat-tracker." We could achieve this in multiple ways, the simplest being to train our own classifie with the classes "squatting" and "not squatting." This would clearly involve building a dataset with thousands of pictures of people either squatting or not squatting.

A much better way to accomplish this would involve running PoseNet (which tracks body joints, as we saw in Chapter 10) on, say, Google Coral USB Accelerator. With its 10-plus FPS, we could track how many times the hip points drop low enough and count much more accurately. All we would need is a Raspberry Pi, a Pi Camera, a Coral USB Accelerator, and a public transport operator to provide a ticket printer and endless free tickets, and we could start making our cities fitter than ever.

r

Figure 15-13. Squatting for tickets

s

Figure 15-13. Squatting for tickets

Cucumber Sorter

Makoto Koike, the son of cucumber farmers in Japan, observed that his parents spent a lot of time postharvest sorting through cucumbers. The sorting required separating cucumbers into nine categories based on fine-grained analysis of very small features such as minute texture differences, tiny scratches, and prickles, along with larger attributes such as size, thickness, and curvature. The system was complicated and hiring part-time workers was pretty much a no-go because of how long it would take to train them. He couldn't sit by idly watching his parents go through the labor-intensive task for more than eight hours every single day.

One detail we didn't mention yet is that Mr. Koike was formerly an embedded systems designer at Toyota. He realized that a lot of the visual inspection could be automated using the power of deep learning. He took more than 7,000 pictures of different cucumbers that his mother had manually sorted over a three-month period. He then trained a classifier that could look at a picture of a cucumber and predict with a high degree of accuracy to which category it belonged. But a classifier is not going to do much on its own now, will it?

Using his experience in embedded systems, he deployed a classifier to a Raspberry Pi that was connected to a camera, a conveyor-belt system, and a sorting arm that pushed each cucumber into one of the multiple bins based on the predictions from the Raspberry Pi (Figure 15-14). The Raspberry Pi ran a small cucumber/not-cucumber classifier as a first pass. For images classified as cucumber, the Raspberry Pi would send the image to a server that ran the more sophisticated multiple category cucumber classifier. Keep in mind that this was in 2015, shortly after TensorFlow was released (long before the days of TensorFlow Lite, MobileNet, and the extra power of the current Raspberry Pi 4 or even 3). With a machine like this, his parents could spend more time in actual farming rather than sorting through already harvested cucumbers for days on end.

Figure 15-14. Makoto Koike's cucumber sorting machine (image source)

Even though the debate on AI taking over jobs appears daily in the news, Mr. Koike's story truly highlights the real value that AI provides: making humans more productive and augmenting our abilities, along with making his parents proud.

Figure 15-14. Makoto Koike's cucumber sorting machine (image source)

Even though the debate on AI taking over jobs appears daily in the news, Mr. Koike's story truly highlights the real value that AI provides: making humans more productive and augmenting our abilities, along with making his parents proud.

Further Exploration

There are two equally important aspects to focus on when becoming a maker: the hardware and the software. One is incomplete without the other. There are more DIY projects to explore than one can build in a lifetime. But the key is to find some that you're passionate about, build them end to end, and feel the excitement when they start to actually function. Gradually, you'll develop the know-how so that the next time you read a project, you can guess how to build it yourself.

The best way to get started is to get hands-on and build more and more projects. Some excellent sources for inspiration include *Hackster.io*, *Hackaday.io*, and *Instructables.com*, which feature a large range of project themes, platforms, and skill levels (from beginner to advanced), often along with the step-by-step instructions.

To reduce the overhead of assembling hardware for beginners, there are several kits available in the market. For example, AI Kits for Raspberry Pi including AIY Vision Kit (Google), AIY Speech Kit (Google), GoPiGo (Dexter Industries), and DuckieTown platform (MIT), to name a few. They lower the barriers for getting started with electronic hardware projects, making the new area more approachable and hence usable for everyone, from high schoolers to rapid prototypers.

On the software side of things, running AI faster on low-power devices means making the models more performant. Model compression techniques like quantization and pruning (as mentioned in Chapter 6) can help get us there. To go even faster (at some loss in accuracy), BNNs, which represent models in one bit instead of the normal 32 bits, are a potential solution. Companies like XNOR.ai and Microsoft's Embedded Learning Library allow us to make such models. For running AI on even lower-power devices like microcontrollers with less than 100 KB of memory, Pete Warden's book *TinyML* is an excellent learning resource.

t

Summary

In this chapter, we introduced some well-known embedded AI devices and looked at how to run a Keras model on some of them. We presented a high-level view of their inner workings, and how they achieve the computing capacity needed for running neural networks. Eventually benchmarking them to develop an intuition on which might be suitable for our projects depending on size, costs, latency, and power requirements. Going from platforms to robotic projects, we looked at some ways in which makers around the world are using them to build electronic projects. The devices discussed in this chapter are obviously just a handful of those available. New devices will keep popping up as the edge becomes a more desirable place to be for AI.

With the power of edge AI, makers can imagine and bring their dream projects to reality, which might have been considered science fiction just a few years ago On your journey in this DIY AI world, remember, you are among a tight-knit community of tinkerers with a lot of openness and willingness to share. The forums are active, so if you ever get stuck on a problem, there are helpful souls to call upon. And hopefully, with your project, you can inspire others to become makers themselves.

Summary

In this chapter, we introduced some well-known embedded AI devices and looked at how to run a Keras model on some of them. We presented a high-level view of their inner workings, and how they achieve the computing capacity needed for running neural networks. Eventually benchmarking them to develop an intuition on which might be suitable for our projects depending on size, costs, latency, and power requirements. Going from platforms to robotic projects, we looked at some ways in which makers around the world are using them to build electronic projects. The devices discussed in this chapter are obviously just a handful of those available. New devices will keep popping up as the edge becomes a more desirable place to be for AI.

With the power of edge AI, makers can imagine and bring their dream projects to reality, which might have been considered science fiction just a few years ago. On your journey in this DIY AI world, remember, you are among a tight-knit community of tinkerers with a lot of openness and willingness to share. The forums are active, so if you ever get stuck on a problem, there are helpful souls to call upon. And hopefully, with your project, you can inspire others to become makers themselves.

Chapter 16. Simulating a Self-Driving Car Using End-to-End Deep Learning with Keras

Contributed by guest authors: Aditya Sharma and Mitchell Spryn

From the Batmobile to Knightrider, from robotaxis to autonomous pizza delivery, self-driving cars have captured the imagination of both modern pop culture and the mainstream media. And why wouldn't they? How often does it happen that a science-fiction concept is brought to life? More important, autonomous cars promise to address some key challenges urban societies are facing today: road accidents, pollution levels, cities becoming increasingly more congested, hours of productivity lost in traffic, and the list goes on.

It should come as no surprise that a full self-driving system is quite complex to build, and cannot be tackled in one book chapter. Much like an onion, any complex problem contains layers that can be peeled. In our case, we intend to tackle one fundamental problem here: how to steer. Even better, we don't need a real car to do this. We will be training a neural network to drive our car autonomously within a simulated environment, using realistic physics and visuals.

But first, a brief bit of history.

SAE LEVELS OF DRIVING AUTOMATION

"Autonomous driving" is a broad term that is often used to describe everything from Adaptive Cruise Control (ACC) technology on your Chrysler 200 to the steering wheel-free cars of the future. To bring some structure around the definition of autonomous driving (or the term more popular in Europe: automated driving), the Society of Automotive Engineers (SAE) International proposed to divide automation capabilities of driving vehicles into six levels, ranging from "No Automation" at Level 0 to "Full Automation" at Level 5. Figure 16-1 gives an overview of these. These

levels are somewhat subject to interpretation, but they give the industry a reference point when talking about autonomous driving. ACC is an example of Level 1 automation, whereas Tesla's cars fall somewhere between Levels 2 and 3. Although there are many startups, technology companies, and automakers working on Level 4 cars, we haven't seen any in production yet. It will take us a while before we realize our dream of a truly fully autonomous Level 5 vehicle.

Figure 16-1. The SAE Levels of Driving Automation (image source)

levels are somewhat subject to interpretation, but they give the industry a reference point when talking about autonomous driving. ACC is an example of Level 1 automation, whereas Tesla's cars fall somewhere between Levels 2 and 3. Although there are many startups, technology companies, and automakers working on Level 4 cars, we haven't seen any in production yet. It will take us a while before we realize our dream of a truly fully autonomous Level 5 vehicle.

Figure 16-1. The SAE Levels of Driving Automation (image source)

A Brief History of Autonomous Driving

Even though it might seem like autonomous driving is a very recent technological development, research in this field originated more than 30 years ago. Scientists at Carnegie Mellon University first tried their hands at this seemingly impossible feat back in 1984 when they started working on what would be known as the Navlab 1. “Navlab” was a shorthand for Carnegie Mellon’s Navigation Laboratory, a not-so-fancy name for what would turn out to be the first step into a very impressive future for humanity. The Navlab 1 was a Chevrolet panel van (Figure 16-2) with racks of hardware onboard that included, among other things, workstations for the research team and a Sun Microsystems supercomputer. It was considered cutting edge at the time, a self-contained computing system on wheels. Long-distance wireless communication was pretty much nonexistent back then, and cloud computing was not even a concept. Putting the entire research team in the back of a highly experimental self-driving van might seem like a dangerous idea, but the Navlab had a maximum speed of 20 MPH and navigated on only empty roads. This ensured the safety of the scientists working onboard. What is very impressive to note though, is that the technology used in this car laid the foundation for technology used in the self-driving cars of today. For example, Navlab 1 used video cameras and a rangefinder (an early version of lidar) to observe its surroundings. For navigation, they used a single-layer neural network to predict steering angles given sensor data. Pretty neat isn’t it?

Figure 16-2. The NavLab 1 in all its glory (image source)

TALK ABOUT YOUR ROBO-ACHIEVERS

There were many more versions of the Navlab 1. Navlab 5 was actually able to steer itself all the way from Pittsburgh to San Diego in 1995, driving all but 50 of the 2,850 mile journey by itself. For this achievement, Navlab 5 was inducted into the Robot Hall of Fame. In the world of robots, being in the Hall of Fame pretty much makes you elite royalty. If you are ever in Pittsburgh and want to meet some other members of this prestigious class, be sure to check out the Roboworld exhibit at the Carnegie Science Center.

TALK ABOUT YOUR ROBO-ACHIEVERS

There were many more versions of the Navlab 1. Navlab 5 was actually able to steer itself all the way from Pittsburgh to San Diego in 1995, driving all but 50 of the 2,850 mile journey by itself. For this achievement, Navlab 5 was inducted into the Robot Hall of Fame. In the world of robots, being in the Hall of Fame pretty much makes you elite royalty. If you are ever in Pittsburgh and want to meet some other members of this prestigious class, be sure to check out the Roboworld exhibit at the Carnegie Science Center.

Deep Learning, Autonomous Driving, and the Data Problem

Even 35 years ago scientists knew that neural networks were going to play a key role in making self-driving cars a reality. Back then, of course, we did not have the technology needed (GPUs, cloud computing, FPGAs, etc.) to train and deploy deep neural networks at scale for autonomous driving to become a reality. Today's autonomous cars use deep learning to perform all kinds of tasks. Table 16-1 lists some examples.

Table 16-1. Examples of deep learning applications in self-driving cars

Task	Examples
Perception	Detect drivable area, lane markings, intersections, crosswalks, etc.
	Detect other vehicles, pedestrians, animals, objects on the road, etc.
	Detect traffic signs and signals
Navigation	Predict and output steering angle, throttle, etc., based on sensor input
	Perform overtaking, lane changes, U-turns, etc.
	Perform merges, exits, navigate roundabouts, etc.
	Drive in tunnels, over bridges, etc.
	Respond to traffic rule breakers, wrong-way drivers, unexpected environment changes, etc.
	Navigate stop-and-go traffic

We know that deep learning requires data. In fact, a general rule of thumb is that more data ensures better model performance. In previous chapters, we saw that training image classifiers can require millions of images. Imagine how much data it would take to train a car to drive itself. Things are also very different with self-driving cars when it comes to the data needed for validation and testing. Not only must a car be able to drive, it must do it safely. Testing and validating model performance is therefore fundamentally critical and requires a lot more data than what is needed for training, unlike traditional deep learning problems. However, it is difficult to accurately predict the exact amount of data needed, and estimates vary. A 2016 study shows that it would take 11 billion miles of

driving for a car to become as good as a human driver. For a fleet of 100 self-driving cars collecting data 24 hours per day, 365 days per year at an average speed of 25 miles per hour, this would take more than 500 years!

Now, of course, it is highly impractical and costly to collect 11 billion miles of data by having fleets of cars drive around in the real world. This is why almost everyone working in this space—be it a large automaker or a startup—uses simulators to collect data and validate trained models. Waymo (Google's self-driving car team), for example, had driven nearly 10 million miles on the road as of late 2018. Even though this is more than any other company on the planet it represents less than one percent of our 11-billion-mile figure. On the other hand, Waymo has driven seven billion miles in simulation. Although everyone uses simulation for validation, some companies build their own simulation tools, whereas others license them from companies like Cognata, Applied Intuition, and AVSimulation. There are some great open source simulation tools available as well: AirSim from Microsoft, Carla from Intel, and Apollo simulation from Baidu. Thanks to these tools, we don't need to connect the CAN bus of our car to learn the science behind making it drive itself.

In this chapter, we use a custom-built version of AirSim, built specifically for Microsoft's Autonomous Driving Cookbook.

SIMULATION AND THE AUTONOMOUS DRIVING INDUSTRY

Simulation plays a very important role in autonomous driving. Before the autonomy stack can be safely deployed on real-world vehicles, it must go through hours and hours of testing and validation inside simulated worlds that accurately represent the environment these vehicles will be driving in. Simulators enable closed-loop validation of algorithms. The ego vehicle in simulation receives sensor data and performs actions based on it, which has a direct affect on the vehicle's environment and influences the next set of sensor readings; hence, closing the loop.

Different types of simulators are utilized based on which part of the autonomy stack is being tested (Figure 16-3). For tasks that require a visually realistic representation of the environment, like testing the perception stack, we use photorealistic simulators. These simulators also try to accurately mimic sensor physics, material dynamics, and so on. AirSim

and Carla are examples of open source photo-realistic simulators. On the other hand, for tasks that can do without realistic visuals, we can save a lot of GPU resources by employing post-perception simulators. Most of these tasks utilize the results of a perception system to do path planning and navigation. Baidu's Apollo is an example of an open source simulator in this category.

Figure 16-3. AirSim's photo realistic simulation versus Apollo postperception simulation (image source)

Cognata, an autonomous driving simulation startup based in Israel, is a leader in the space with its cloud-scale, photo-realistic simulation platform. Cognata is addressing some of the key challenges in this industry using a very innovative approach that uses DNNs. "I came to this field after working on computer vision–based ADAS [Advanced Driver Assistance Systems] applications for close to a decade," said Cognata CEO and founder Danny Atsmon. "I found out that using OpenGL-based rendering techniques does not yield the feature quality which is needed to train or validate a deep neural network and came to a decision that a new rendering technique is needed for that. Our rendering engine uses DNNs to learn distributions from real recorded data and renders the simulated data in a way that mimics the actual sensor."

We are indeed living in a fantastic time in which hardware and software innovations are coming together like never before to solve one of the most complex technological challenges we have ever seen: the self-driving car.

and Carla are examples of open source photo-realistic simulators. On the other hand, for tasks that can do without realistic visuals, we can save a lot of GPU resources by employing post-perception simulators. Most of these tasks utilize the results of a perception system to do path planning and navigation. Baidu's Apollo is an example of an open source simulator in this category.

Figure 16-3. AirSim's photo realistic simulation versus Apollo postperception simulation (image source)

Cognata, an autonomous driving simulation startup based in Israel, is a leader in the space with its cloud-scale, photo-realistic simulation platform. Cognata is addressing some of the key challenges in this industry using a very innovative approach that uses DNNs. "I came to this field after working on computer vision–based ADAS [Advanced Driver Assistance Systems] applications for close to a decade," said Cognata CEO and founder Danny Atsmon. "I found out that using OpenGL-based rendering techniques does not yield the feature quality which is needed to train or validate a deep neural network and came to a decision that a new rendering technique is needed for that. Our rendering engine uses DNNs to learn distributions from real recorded data and renders the simulated data in a way that mimics the actual sensor."

We are indeed living in a fantastic time in which hardware and software innovations are coming together like never before to solve one of the most complex technological challenges we have ever seen: the self-driving car.

The “Hello, World!” of Autonomous Driving: Steering Through a Simulated Environment

In this section, we implement the “Hello, World!” problem of autonomous driving. Self-driving cars are complex, with dozens of sensors streaming gigabytes of data and multiple decisions being made while driving down a road. Much like programming, for the “Hello, World!” of self-driving cars, we strip the requirements down to basic fundamentals:

- The car always stays on the road.
- For external sensor input, the car uses a single image frame from a single camera mounted on the front of the hood. No other sensors (lidar, radar, etc.) are used.
- Based on this single-image input, going at a constant speed, the car predicts its output steering angle. No other possible outputs (brake, throttle, gear changes, etc.) are predicted.
- There are no other vehicles, pedestrians, animals, or anything else on the road or relevant surroundings.
- The road being driven on is single lane, with no markings or traffic signs and signals and no rules for staying on the left or right side of the road.
- The road mostly changes in terms of turns (which will require changing steering angle to navigate) and not in terms of elevation (which would require a change in throttle, applying brakes, etc.).

The “Hello, World!” of Autonomous Driving: Steering Through a Simulated Environment

In this section, we implement the “Hello, World!” problem of autonomous driving. Self-driving cars are complex, with dozens of sensors streaming gigabytes of data and multiple decisions being made while driving down a road. Much like programming, for the “Hello, World!” of self-driving cars, we strip the requirements down to basic fundamentals:

- The car always stays on the road.
- For external sensor input, the car uses a single image frame from a single camera mounted on the front of the hood. No other sensors (lidar, radar, etc.) are used.
- Based on this single-image input, going at a constant speed, the car predicts its output steering angle. No other possible outputs (brake, throttle, gear changes, etc.) are predicted.
- There are no other vehicles, pedestrians, animals, or anything else on the road or relevant surroundings.
- The road being driven on is single lane, with no markings or traffic signs and signals and no rules for staying on the left or right side of the road.
- The road mostly changes in terms of turns (which will require changing steering angle to navigate) and not in terms of elevation (which would require a change in throttle, applying brakes, etc.).

Setup and Requirements

We will be implementing the "Hello, World!" problem in AirSim's Landscape map (Figure 16-4). AirSim is an open source photo-realistic simulation platform and is a popular research tool for training data collection and validation of deep learning–based autonomous system model development.

Figure 16-4. The Landscape map in AirSim

You can find the code for this chapter in the end-to-end deep learning tutorial from the *Autonomous Driving Cookbook* on GitHub in the form of Jupyter Notebooks. In the cookbook repository, go to *AirSimE2EDeepLearning/*. We car do so by running the following:

```
$ git clone https://github.com/Microsoft/AutonomousDrivingCookbook.git
$ cd AutonomousDrivingCookbook/AirSimE2EDeepLearning/
```

We will use Keras, a library with which we are already familiar, to implement our neural network. It's not necessary for us to learn any new deep learning concepts to work on this problem other than what has already been introduced in prior chapters of this book.

For this chapter, we have created a dataset by driving around in AirSim. The uncompressed size of the dataset is 3.25 GB. This is a little larger than datasets we've been using, but remember, we are implementing the "Hello, World!" of a

highly complex problem. For comparison, it is fairly normal practice for automakers to collect multiple petabytes of data on the road every day. You can download the dataset by going to *aka.ms/AirSimTutorialDataset*.

We can run the provided code and train our model on a Windows or Linux machine. We created a standalone build of AirSim for this chapter, which you can download from *aka.ms/ADCookbookAirSimPackage*. After you've downloaded it, extract the simulator package to the location of your choice. Take note of the path because you will need it later. Please note that this build runs only on Windows but is needed only to see our final trained model in action. If you're using Linux, you can take the model files and run them on a Windows machine with the provided simulator package.

AirSim is a highly photo-realistic environment, which means it is capable of generating lifelike pictures of the environment for the model to train on. You may have encountered similar graphics while playing high-definition video games. Given the size of the data and the graphic quality of the provided simulator package, a GPU is certainly preferable for running the code as well as the simulator.

Finally, there are some additional tools and Python dependencies needed to run the code provided. You can find the details for these at Environment Setup in the README file in the code repository. At a high level, we need Anaconda with Python 3.5 (or higher), TensorFlow (to run as a backend to Keras), and h5py (for storing and reading data and model files). After we have our Anaconda environment set up, we can install the needed Python dependencies by running the *InstallPackages.py* file as root or administrator.

Table 16-2 provides a summary of all the requirements that we just defined.

Table 16-2. Setup and requirements summary

Item	Requirements/Link	Notes/Comments
Code repository	*https://oreil.ly/_IJl9*	Can be run on a Windows or Linux machine
Jupyter Notebooks used	*DataExplorationAndPreparation.ipynb* *TrainModel.ipynb* *TestModel.ipynb*	

Dataset download	*aka.ms/AirSimTutorialDataset*	3.25 GB in size; needed to train model
Simulator download	*aka.ms/ADCookbookAirSimPackage*	Not needed to train model, only to deploy and run it; runs only on Windows
GPU	Recommended for training, required for running simulator	
Other tools + Python dependencies	Environment Setup section in README file in the repository	

With all that taken care of, let's get started!

Dataset download	*aka.ms/AirSimTutorialDataset*	3.25 GB in size; needed to train model
Simulator download	*aka.ms/ADCookbookAirSimPackage*	Not needed to train model, only to deploy and run it; runs only on Windows
GPU	Recommended for training, required for running simulator	
Other tools + Python dependencies	Environment Setup section in README file in the repository	

With all that taken care of, let's get started!

Data Exploration and Preparation

As we journey deeper into the world of neural networks and deep learning, you will notice that much of the magic of machine learning doesn't come from how a neural network is built and trained, but from the data scientist's understanding of the problem, the data, and the domain. Autonomous driving is no different. As you will see shortly, having a deep understanding of the data and the problem we are trying to solve is key when teaching our car how to drive itself.

All the steps here are also detailed in the Jupyter Notebook *DataExplorationAndPreparation.ipynb*. We start by importing all the necessary modules for this part of the exercise:

```
import os
import random

import numpy as np
import pandas as pd
import h5py

import matplotlib.pyplot as plt
from PIL import Image, ImageDraw

import Cooking #This module contains helper code. Please do not edit this file.
```

Next, we make sure our program can find the location of our unzipped dataset. We also provide it a location to store our processed (or "cooked") data:

```
# << Point this to the directory containing the raw data >>
RAW_DATA_DIR = 'data_raw/'

# << Point this to the desired output directory for the cooked (.h5) data >>
COOKED_DATA_DIR = 'data_cooked/'

# The folders to search for data under RAW_DATA_DIR
# For example, the first folder searched will be RAW_DATA_DIR/normal_1
DATA_FOLDERS = ['normal_1', 'normal_2', 'normal_3', 'normal_4', 'normal_5',
        'normal_6', 'swerve_1', 'swerve_2', 'swerve_3']

# The size of the figures and illustrations used
FIGURE_SIZE = (10,10)
```

Looking at the provided dataset we will see that it contains nine folders: *normal_1* through *normal_6* and *swerve_1* through *swerve_3*. We come back to the naming of these folders later. Within each of these folders are images as well as *.tsv* (or *.txt*) files. Let's check out one of these images:

```
sample_image_path = os.path.join(RAW_DATA_DIR, 'normal_1/images/img_0.png')
sample_image = Image.open(sample_image_path)
plt.title('Sample Image')
plt.imshow(sample_image)
plt.show()
```

We should see the following output. We can see that this image was captured by a camera placed centrally on the hood of the car (which is slightly visible at the bottom in Figure 16-5). We can also see the Landscape environment used for this problem from the simulator. Notice that no other vehicles or objects are on the road, keeping in line with our requirements. Although the road appears to be snowy, for our experiment, the snow is only visual and not physical; that is, it will have no effect on the physics and movement of our car. Of course, in the real world, it is critical to accurately replicate the physics of snow, rain, etc., so that our simulators can produce highly accurate versions of reality. This is a very complicated task to undertake and many companies dedicate a lot of resources to it. Thankfully for our purpose, we can safely ignore all that and treat the snow as just white pixels in our image.

Figure 16-5. Plot showing the contents of the file normal_1/images/img_0.png

Let's also get a preview of what the rest of the data looks like by displaying the contents of one of the *.tsv/.txt* files:

```
sample_tsv_path = os.path.join(RAW_DATA_DIR, 'normal_1/airsim_rec.txt')
sample_tsv = pd.read_csv(sample_tsv_path, sep='\t')
sample_tsv.head()
```

We should see the following output:

	Timestamp	Speed (kmph)	Throttle	Steering	Brake	Gear	ImageName
0	93683464	0	0.0	0.000000	0.0	N	img_0.png
1	93689595	0	0.0	0.000000	0.0	N	img_1.png
2	93689624	0	0.0	-0.035522	0.0	N	img_2.png
3	93689624	0	0.0	-0.035522	0.0	N	img_3.png
4	93689624	0	0.0	-0.035522	0.0	N	img_4.png

There is a lot going on here, so let's break it down. First, we can see that each row in the table corresponds to an image frame (we are displaying only the first five frames here). Because this is at the beginning of our data collection drive, we can see that the car hasn't started moving yet: the speed and throttle are 0 and the gear is neutral. Because we are trying to predict only steering angles for our problem, we won't be predicting the speed, throttle, brake, or gear values. These values will be used, however, as part of the input data (more on this later).

We should see the following output:

	Timestamp	Speed (kmph)	Throttle	Steering	Brake	Gear	ImageName
0	93683464	0	0.0	0.000000	0.0	N	img_0.png
1	93689595	0	0.0	0.000000	0.0	N	img_1.png
2	93689624	0	0.0	-0.035522	0.0	N	img_2.png
3	93689624	0	0.0	-0.035522	0.0	N	img_3.png
4	93689624	0	0.0	-0.035522	0.0	N	img_4.png

There is a lot going on here, so let’s break it down. First, we can see that each row in the table corresponds to an image frame (we are displaying only the first five frames here). Because this is at the beginning of our data collection drive, we can see that the car hasn’t started moving yet: the speed and throttle are 0 and the gear is neutral. Because we are trying to predict only steering angles for our problem, we won’t be predicting the speed, throttle, brake, or gear values. These values will be used, however, as part of the input data (more on this later).

Identifying the Region of Interest

Let's get back to our sample image now. All the observations we made so far were from the eyes of a human being. For a moment, let's take a look at the image again, only this time we step into the shoes (or tires) of a car that is just beginning to learn how to drive itself and is trying to understand what it is seeing.

If I'm the car, looking at this image that my camera presented me, I can divide it into three distinct parts (Figure 16-6). First, there is the lower third, which more or less looks consistent. It is made up of mostly straight lines (the road, lane divider, road fences, etc.) and has a uniform coloring of white, black, gray, and brown. There is also a weird black arc at the very bottom (this is the hood of the car). The upper third of the image, like the lower third, is also consistent. It is mostly gray and white (the sky and the clouds). Lastly, there is the middle third and it has a lot going on. There are big brown and grey shapes (mountains), which are not uniform at all. There are also four tall green figures (trees), their shape and color are different from anything else I see, so they must be super important. As I am presented more images, the upper third and the lower third don't change all that much but the middle third undergoes a lot of change. Hence, I conclude that this is the part that I should be focusing on the most.

Figure 16-6. The three parts of the image as seen by the self-driving car

Do you see the challenge in front of our car? It has no way of knowing what part

of the image being presented to it is important. It might try to associate the changing steering angles with the changing scenery instead of focusing on turns in the road, which are very subtle compared to everything else that is going on. Not only are the changes in the scenery confusing, but they are also not relevant to the problem we are trying to solve here at all. The sky above, although mostly unchanging, also does not provide us with any information relevant to the steering of our car. Finally, the part of the hood being captured in every picture is also nonrelevant.

Let us fix this by focusing only on the relevant portion of the image. We do this by creating a region of interest (ROI) in our image. As we can imagine, there is no hard-and-fast rule to dictate what our ROI should look like. It really depends on our use case. For us, because the camera is in a fixed position and there are no elevation changes on the road, the ROI is a simple rectangle focusing on the road and road boundaries. We can see the ROI by running the following code snippet:

```
sample_image_roi = sample_image.copy()

fillcolor=(255,0,0)
draw = ImageDraw.Draw(sample_image_roi)
points = [(1,76), (1,135), (255,135), (255,76)]
for i in range(0, len(points), 1):
    draw.line([points[i], points[(i+1)%len(points)]], fill=fillcolor, width=3)
del draw

plt.title('Image with sample ROI')
plt.imshow(sample_image_roi)
plt.show()
```

We should see the output shown in Figure 16-7.

Figure 16-7. The ROI for our car to focus on during training

Our model will focus only on the road now and not be confused by anything else going on in the environment. This also reduces the size of the image by half, which will make training our neural network easier.

Figure 16-7. The ROI for our car to focus on during training

Our model will focus only on the road now and not be confused by anything else going on in the environment. This also reduces the size of the image by half, which will make training our neural network easier.

Data Augmentation

As mentioned earlier, it is always better to have as much data as we can get our hands on while training deep learning models. We have seen in previous chapters the importance of data augmentation and how it helps to not only get more training data but to also avoid overfitting. Most data augmentation techniques discussed previously for image classification, however, will not be useful for our current autonomous driving problem. Let's take the example of rotation. Rotating images randomly by 20 degrees is very helpful when training a classifier for a smartphone camera, but the camera on the hood of our car is fixed in place and will never see a rotated image (unless our car is doing flips, at which point we have larger concerns). The same goes for random shifts; we will never encounter those while driving. There is, however, something else we can do to augment the data for our current problem.

Looking closely at our image, we can observe that flipping it on the y-axis produces an image that could have easily come from a different test run (Figure 16-8). We can effectively double the size of our dataset if we used the flipped versions of images along with their regular ones. Flipping an image this way will require us to modify the steering angle associated with it as well. Because our new image is a reflection of our original, we can just change the sign of the corresponding steering angle (e.g., from 0.212 to –0.212).

Figure 16-8. Flipping image on the y-axis

There is another thing we can do to augment our data. Autonomous cars need to be ready for not only changes on the roads, but also for changes in external conditions like weather, time of day, and available light. Most simulators available today allow us to create these conditions synthetically. All of our data

was collected in bright lighting conditions. Instead of going back to the simulator to collect more data under a variety of lighting conditions, we could just introduce random lighting changes by adjusting image brightness while training on the data we have. Figure 16-9 shows an example.

Figure 16-9. Reducing image brightness by 40%

was collected in bright lighting conditions. Instead of going back to the simulator to collect more data under a variety of lighting conditions, we could just introduce random lighting changes by adjusting image brightness while training on the data we have. Figure 16-9 shows an example.

Figure 16-9. Reducing image brightness by 40%

Dataset Imbalance and Driving Strategies

Deep learning models are only as good as the data on which they are trained. Unlike humans, the AI of today cannot think and reason for itself. Hence, when presented with a completely new situation it will fall back on what it has seen before and make predictions based on the dataset it was trained on. Additionally, the statistical nature of deep learning makes it ignore instances that happen infrequently as aberrations and outliers, even if they are of significance. This is called *dataset imbalance*, and it is a common nuisance for data scientists.

Imagine that we are trying to train a classifier that looks at dermatoscopic images to detect whether a lesion is benign or malignant. Suppose that our dataset has one million images, 10,000 of which contain lesions that are malignant. It is very likely that the classifier we train on this data always predicts images to be benign and never malignant. This happens because deep learning algorithms are designed to minimize error (or maximize accuracy) while training. By classifying all images as benign, our model attains 99% accuracy and calls it a day, while severely failing at the job it was trained to perform: detecting cancerous lesions. This problem is usually solved by training predictors on a more balanced dataset.

Autonomous driving is not immune to the problem of dataset imbalance. Let's take our steering-angle prediction model. What do we know about steering angles from our day-to-day driving experience? While driving down a road we mostly drive straight. If our model was trained on only normal driving data, it would never learn how to navigate turns due to their relative scarcity in the training dataset. To solve this, it is important that our dataset contains data not only for a normal driving strategy but also for a "swerved" driving strategy in a statistically significant amount. To illustrate what we mean by this, let's go back to our *.tsv*/*.txt* files. Earlier in the chapter, we pointed out the naming of our data folders. It should now be clear that our dataset has data from six collection runs using a normal driving strategy and from three collection runs using a swerve driving strategy.

Let us aggregate the data from all *.tsv*/*.txt* files into a single DataFrame to make the analysis easier:

```
full_path_raw_folders = [os.path.join(RAW_DATA_DIR, f) for f in DATA_FOLDERS]

dataframes = []
for folder in full_path_raw_folders:
    current_dataframe = pd.read_csv(os.path.join(folder, 'airsim_rec.txt'),
                                    sep='\t')
    current_dataframe['Folder'] = folder
    dataframes.append(current_dataframe)
dataset = pd.concat(dataframes, axis=0)
```

Let's plot some steering angles from both driving strategies on a scatter plot:

```
min_index = 100
max_index = 1500
steering_angles_normal_1 = dataset[dataset['Folder'].apply(lambda v: 'normal_1'
in v)]['Steering'][min_index:max_index]
steering_angles_swerve_1 = dataset[dataset['Folder'].apply(lambda v: 'swerve_1'
in v)]['Steering'][min_index:max_index]

plot_index = [i for i in range(min_index, max_index, 1)]
fig = plt.figure(figsize=FIGURE_SIZE)
ax1 = fig.add_subplot(111)
ax1.scatter(plot_index, steering_angles_normal_1, c='b', marker='o',
label='normal_1')
ax1.scatter(plot_index, steering_angles_swerve_1, c='r', marker='_',
label='swerve_1')
plt.legend(loc='upper left');
plt.title('Steering Angles for normal_1 and swerve_1 runs')
plt.xlabel('Time')
plt.ylabel('Steering Angle')
plt.show()
```

Figure 16-10 shows the results.

Figure 16-10. Plot showing steering angles from the two driving strategies

Let's also plot the number of data points collected with each strategy:

```
dataset['Is Swerve'] = dataset.apply(lambda r: 'swerve' in r['Folder'], axis=1)
grouped = dataset.groupby(by=['Is Swerve']).size().reset_index()
grouped.columns = ['Is Swerve', 'Count']

def make_autopct(values):
    def my_autopct(percent):
        total = sum(values)
        val = int(round(percent*total/100.0))
        return '{0:.2f}% ({1:d})'.format(percent,val)
    return my_autopct

pie_labels = ['Normal', 'Swerve']
fig, ax = plt.subplots(figsize=FIGURE_SIZE)
ax.pie(grouped['Count'], labels=pie_labels, autopct =
make_autopct(grouped['Count']), explode=[0.1, 1], textprops={'weight': 'bold'},
colors=['lightblue', 'salmon'])
plt.title('Number of data points per driving strategy')
plt.show()
```

Figure 16-11 shows those results.

Figure 16-11. Dataset split for the two driving strategies

Looking at Figure 16-10, as we had expected, we see that the normal driving strategy produces steering angles that we would observe during everyday driving, mostly straight on the road with the occasional turn. By contrast, the swerve driving strategy mostly focuses on sharp turns and hence has higher values for steering angles. As shown in Figure 16-11, a combination of these two strategies gives us a nice, albeit unrealistic in real life, distribution to train on with a 75/25 split. This also further solidifies the importance of simulation for autonomous driving because it is unlikely we'd be collecting data using the swerve strategy in real life with an actual car.

Before closing our discussion on data preprocessing and dataset imbalance, let's take a final look at the distribution of our steering angles for the two driving strategies by plotting them on a histogram (Figure 16-12):

```
bins = np.arange(-1, 1.05, 0.05)
normal_labels = dataset[dataset['Is Swerve'] == False]['Steering']
swerve_labels = dataset[dataset['Is Swerve'] == True]['Steering']

def steering_histogram(hist_labels, title, color):
    plt.figure(figsize=FIGURE_SIZE)
    n, b, p = plt.hist(hist_labels.as_matrix(), bins, normed=1, facecolor=color)
    plt.xlabel('Steering Angle')
```

```
    plt.ylabel('Normalized Frequency')
    plt.title(title)
    plt.show()

steering_histogram(normal_labels, 'Normal driving strategy label distribution',
'g')
steering_histogram(swerve_labels, 'Swerve driving strategy label distribution',
'r')
```

Figure 16-12. Steering angle distribution for the two driving strategies

As noted earlier, the swerve driving strategy gives us a much broader range of angles compared to the normal driving strategy, as shown in Figure 16-12. These angles will help our neural network react appropriately in case it ever finds the car going off the road. The problem of imbalance in our dataset is somewhat solved, but not entirely. We still have a lot of zeros, across both driving strategies. To balance our dataset further, we could just ignore a portion of these during training. Although this gives us a very balanced dataset, it greatly reduces the overall number of data points available to us. We need to keep this in mind when we build and train our neural network.

Before moving on, let’s make sure our data is suitable for training. Let’s take the raw data from all folders, split it into train, test, and validation datasets, and compress them into HDF5 files. The HDF5 format allows us to access data in chunks without reading the entire dataset into memory at once. This makes it ideal for deep learning problems. It also works seamlessly with Keras. The following code will take some time to run. When it has finished running, we will

have three dataset files: *train.h5*, *eval.h5*, and *test.h5*.

```
train_eval_test_split = [0.7, 0.2, 0.1]
full_path_raw_folders = [os.path.join(RAW_DATA_DIR, f) for f in DATA_FOLDERS]
Cooking.cook(full_path_raw_folders, COOKED_DATA_DIR, train_eval_test_split)
```

Each dataset file has four parts:

Image

A NumPy array containing the image data.

Previous_state

A NumPy array containing the last known state of the car. This is a (steering, throttle, brake, speed) tuple.

Label

A NumPy array containing the steering angles that we wish to predict (normalized on the range -1..1).

Metadata

A NumPy array containing metadata about the files (which folder they came from, etc.).

We are now ready to take the observations and learnings from our dataset and start training our model.

have three dataset files: *train.h5*, *eval.h5*, and *test.h5*.

```
train_eval_test_split = [0.7, 0.2, 0.1]
full_path_raw_folders = [os.path.join(RAW_DATA_DIR, f) for f in DATA_FOLDERS]
Cooking.cook(full_path_raw_folders, COOKED_DATA_DIR, train_eval_test_split)
```

Each dataset file has four parts:

Image

A NumPy array containing the image data.

Previous_state

A NumPy array containing the last known state of the car. This is a (steering, throttle, brake, speed) tuple.

Label

A NumPy array containing the steering angles that we wish to predict (normalized on the range -1..1).

Metadata

A NumPy array containing metadata about the files (which folder they came from, etc.).

We are now ready to take the observations and learnings from our dataset and start training our model.

Training Our Autonomous Driving Model

All the steps in this section are also detailed in the Jupyter Notebook *TrainModel.ipynb*. As before, we begin by importing some libraries and defining paths:

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense, Lambda,
Input, concatenate
from tensorflow.keras.optimizers import Adam, SGD, Adamax, Nadam
from tensorflow.keras.callbacks import ReduceLROnPlateau, ModelCheckpoint,
        CSVLogger
from tensorflow.keras.callbacks import EarlyStopping
import tensorflow.keras.backend as K
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Sequential, Model, load_model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense,
from tensorflow.keras.layers import Lambda, Input, concatenate, BatchNormalization

from keras_tqdm import TQDMNotebookCallback
import json
import os
import numpy as np
import pandas as pd
from Generator import DriveDataGenerator
from Cooking import checkAndCreateDir
import h5py
from PIL import Image, ImageDraw
import math
import matplotlib.pyplot as plt

# << The directory containing the cooked data from the previous step >>
COOKED_DATA_DIR = 'data_cooked/'
# << The directory in which the model output will be placed >>
MODEL_OUTPUT_DIR = 'model'
```

Let's also set up our datasets:

```
train_dataset = h5py.File(os.path.join(COOKED_DATA_DIR, 'train.h5'), 'r')
eval_dataset = h5py.File(os.path.join(COOKED_DATA_DIR, 'eval.h5'), 'r')
test_dataset = h5py.File(os.path.join(COOKED_DATA_DIR, 'test.h5'), 'r')
```

```
    num_train_examples = train_dataset['image'].shape[0]
    num_eval_examples = eval_dataset['image'].shape[0]
    num_test_examples = test_dataset['image'].shape[0]

    batch_size=32

}
```

```
num_train_examples = train_dataset['image'].shape[0]
num_eval_examples = eval_dataset['image'].shape[0]
num_test_examples = test_dataset['image'].shape[0]

batch_size=32
```

Drive Data Generator

We were introduced to the concept of Keras data generators in previous chapters A data generator iterates through the dataset and reads data in chunks from the disk. This allows us to keep both our CPU and GPU busy, increasing throughput To implement ideas discussed in the previous section we created our own Keras generator called `DriveDataGenerator`.

Let's recap some of our observations from the previous section:

- Our model should focus only on the ROI within each image.
- We can augment our dataset by flipping images horizontally and reversing the sign of the steering angle.
- We can further augment the data by introducing random brightness changes in the images. This will simulate different lighting conditions and make our model more robust.
- We can randomly drop a percentage of data points where the steering angle is zero so that the model sees a balanced dataset when training.
- Our overall number of data points available will be reduced significantly post-balancing.

Let's see how the `DriveDataGenerator` takes care of the first four of these items. We will return to last item when we begin designing our neural network.

The ROI is a simple image crop. [76,135,0,255] (shown in the code block that follows) is the [x1,x2,y1,y2] value of the rectangle representing the ROI. The generator extracts this rectangle from each image. We can use the parameter `roi` to modify the ROI.

The horizontal flip is fairly straightforward. When generating batches, random images are flipped along the y-axis and their steering angle values are reversed i the parameter `horizontal_flip` is set to `True`.

For random brightness changes, we introduce the parameter `brighten_range`. Setting this parameter to `0.4` modifies the brightness of images in any given

batch randomly up to 40%. We do not recommend increasing this beyond `0.4`. To compute brightness, we transform the image from RGB to HSV space, scale the "V" coordinate up or down, and transform back to RGB.

For dataset balancing through dropping zeros, we introduce the parameter `zero_drop_percentage`. Setting this to `0.9` will randomly drop 90% of the 0-label data points in any given batch.

Let's initialize our generator with these parameters:

```
data_generator = DriveDataGenerator(rescale=1./255., horizontal_flip=True,
brighten_range=0.4)


train_generator = data_generator.flow\
    (train_dataset['image'], train_dataset['previous_state'],
    train_dataset['label'], batch_size=batch_size, zero_drop_percentage=0.95,
    roi=[76,135,0,255])
eval_generator = data_generator.flow\
    (eval_dataset['image'], eval_dataset['previous_state'],
    eval_dataset['label'],
batch_size=batch_size, zero_drop_percentage=0.95, roi=[76,135,0,255]
```

Let's visualize some sample data points by drawing labels (steering angle) over their corresponding images:

```
def draw_image_with_label(img, label, prediction=None):
    theta = label * 0.69 #Steering range for the car is +- 40 degrees -> 0.69
# radians
    line_length = 50
    line_thickness = 3
    label_line_color = (255, 0, 0)
    prediction_line_color = (0, 255, 255)
    pil_image = image.array_to_img(img, K.image_data_format(), scale=True)
    print('Actual Steering Angle = {0}'.format(label))
    draw_image = pil_image.copy()
    image_draw = ImageDraw.Draw(draw_image)
    first_point = (int(img.shape[1]/2), img.shape[0])
    second_point = (int((img.shape[1]/2) + (line_length * math.sin(theta))),
int(img.shape[0] - (line_length * math.cos(theta))))
    image_draw.line([first_point, second_point], fill=label_line_color,
width=line_thickness)

    if (prediction is not None):
        print('Predicted Steering Angle = {0}'.format(prediction))
        print('L1 Error: {0}'.format(abs(prediction-label)))
```

```
        theta = prediction * 0.69
        second_point = (int((img.shape[1]/2) + ((line_length/2) *
math.sin(theta))), int(img.shape[0] - ((line_length/2) * math.cos(theta))))
        image_draw.line([first_point, second_point], fill=prediction_line_color,
width=line_thickness * 3)

    del image_draw
    plt.imshow(draw_image)
    plt.show()

[sample_batch_train_data, sample_batch_test_data] = next(train_generator)
for i in range(0, 3, 1):
    draw_image_with_label(sample_batch_train_data[0][i],
        sample_batch_test_data[i])
```

We should see an output similar to Figure 16-13. Observe that we are now looking at only the ROI when training the model, thereby ignoring all the nonrelevant information present in the original images. The line indicates the ground truth steering angle. This is the angle the car was driving at when that image was taken by the camera during data collection.

Figure 16-13. Drawing steering angles on images

```
        theta = prediction * 0.69
        second_point = (int((img.shape[1]/2) + ((line_length/2) *
math.sin(theta))), int(img.shape[0] - ((line_length/2) * math.cos(theta))))
        image_draw.line([first_point, second_point], fill=prediction_line_color,
width=line_thickness * 3)

    del image_draw
    plt.imshow(draw_image)
    plt.show()

[sample_batch_train_data, sample_batch_test_data] = next(train_generator)
for i in range(0, 3, 1):
    draw_image_with_label(sample_batch_train_data[0][i],
        sample_batch_test_data[i])
```

We should see an output similar to Figure 16-13. Observe that we are now looking at only the ROI when training the model, thereby ignoring all the nonrelevant information present in the original images. The line indicates the ground truth steering angle. This is the angle the car was driving at when that image was taken by the camera during data collection.

Figure 16-13. Drawing steering angles on images

Model Definition

We are now ready to define the architecture of our neural network. It is here that we must take into account the problem of our dataset being very limited in size after removing zeros. Because of this limitation, we cannot build a network that is too deep. Because we are dealing with images, we will need a few convolutional/max-pooling pairs to extract features.

Images alone, however, might not be enough to lead our model to convergence. Using only images to train also does not align with how driving decisions are made in the real world. When driving down the road, we are not only perceiving our surrounding environment, we are also aware of how fast we are going, how much we are turning and, the state of our gas and brake pedals. Input from sensors like cameras, lidars, radars, and so on being fed into a neural network corresponds to only one portion of all information a driver has at hand while making a driving decision in the real world. An image presented to our neural network could have been taken from a stationary car or a car driving at 60 mph; the network would have no way of knowing which. Turning the steering wheel by two degrees to the right while driving at 5 mph will yield very different results compared to doing the same at 50 mph. In short, a model trying to predic steering angles should not rely on sensory input alone. It also needs information about the current state of the car. Thankfully for us, we do have this information available.

At the end of the previous section, we noted that our datasets have four parts. For each image, in addition to the steering angle label and metadata, we also recorded the last known state of the car corresponding to the image. This was stored in the form of a (steering, throttle, brake, speed) tuple, and we will use this information, along with our images, as input to the neural network. Note tha this does not violate our “Hello, World!” requirements, because we are still usin the single camera as our only external sensor.

Putting all we discussed together, you can see the neural network that we will be using for this problem in Figure 16-14. We are using three convolutional layers with 16, 32, 32 filters, respectively, and a (3,3) convolutional window. We are merging the image features (output from convolutional layers) with an input layer supplying the previous state of the car. The combined feature set is then

passed through two fully connected layers with 64 and 10 hidden neurons, respectively. The activation function used in our network is ReLU. Notice that unlike the classification problems we have been working on in previous chapters, the last layer of our network is a single neuron with no activation. This is because the problem we are trying to solve is a regression problem. The outpu of our network is the steering angle, a floating-point number, unlike the discrete classes that we were predicting earlier.

t

t
g

t

Figure 16-14. Network architecture

Let's now implement our network. We can use `model.summary()`:

```
image_input_shape = sample_batch_train_data[0].shape[1:]
state_input_shape = sample_batch_train_data[1].shape[1:]
activation = 'relu'

# Create the convolutional stacks
```

```
pic_input = Input(shape=image_input_shape)

img_stack = Conv2D(16, (3, 3), name="convolution0", padding='same',
activation=activation)(pic_input)
img_stack = MaxPooling2D(pool_size=(2,2))(img_stack)
img_stack = Conv2D(32, (3, 3), activation=activation, padding='same',
name='convolution1')(img_stack)
img_stack = MaxPooling2D(pool_size=(2, 2))(img_stack)
img_stack = Conv2D(32, (3, 3), activation=activation, padding='same',
name='convolution2')(img_stack)
img_stack = MaxPooling2D(pool_size=(2, 2))(img_stack)
img_stack = Flatten()(img_stack)
img_stack = Dropout(0.2)(img_stack)

# Inject the state input
state_input = Input(shape=state_input_shape)
merged = concatenate([img_stack, state_input])

# Add a few dense layers to finish the model
merged = Dense(64, activation=activation, name='dense0')(merged)
merged = Dropout(0.2)(merged)
merged = Dense(10, activation=activation, name='dense2')(merged)
merged = Dropout(0.2)(merged)
merged = Dense(1, name='output')(merged)

adam = Nadam(lr=0.0001, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
model = Model(inputs=[pic_input, state_input], outputs=merged)
model.compile(optimizer=adam, loss='mse')
```

Callbacks

One of the nice features of Keras is the ability to declare *callbacks*. Callbacks are functions that are executed after each epoch of training and help us to gain ar insight into the training process as well as control hyperparameters to an extent. They also let us define conditions to perform certain actions while training is underway; for example, stopping the training early if the loss stops decreasing. We will use a few callbacks for our experiment:

`ReduceLROnPlateau`

If the model is near a minimum and the learning rate is too high, the model will circle around that minimum without ever reaching it. This callback will allow the model to reduce its learning rate when the validation loss hits a plateau and stops improving, allowing us to reach the optimal point.

`CSVLogger`

This lets us log the output of the model after each epoch into a CSV file, which will allow us to track the progress without needing to use the console.

`ModelCheckpoint`

Generally, we will want to use the model that has the lowest loss on the validation set. This callback will save the model each time the validation loss improves.

`EarlyStopping`

We will want to stop training when the validation loss stops improving. Otherwise, we risk overfitting. This option will detect when the validation loss stops improving and will stop the training process when that occurs.

Let's now go ahead and implement these callbacks:

```
plateau_callback = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3,
min_lr=0.0001, verbose=1)

checkpoint_filepath = os.path.join(MODEL_OUTPUT_DIR, 'models', '{0}_model.{1}
-{2}.h5'.format('model', '{epoch:02d}', '{val_loss:.7f}'))
checkAndCreateDir(checkpoint_filepath)
checkpoint_callback = ModelCheckpoint(checkpoint_filepath, save_best_only=True,
verbose=1)

csv_callback = CSVLogger(os.path.join(MODEL_OUTPUT_DIR, 'training_log.csv'))

early_stopping_callback = EarlyStopping(monitor='val_loss', patience=10,
                                                            verbose=1)
l
nbcallback = TQDMNotebookCallback()
setattr(nbcallback, 'on_train_batch_begin', lambda x,y: None)
setattr(nbcallback, 'on_train_batch_end', lambda x,y: None)
setattr(nbcallback, 'on_test_begin', lambda x: None)
setattr(nbcallback, 'on_test_end', lambda x: None)
setattr(nbcallback, 'on_test_batch_begin', lambda x,y: None)
setattr(nbcallback, 'on_test_batch_end', lambda x,y: None)

callbacks=[plateau_callback, csv_callback, checkpoint_callback,
early_stopping_callback, nbcallback]
```

We're now all set to kick off training. The model will take a while to train, so this will make for a nice Netflix break. The training process should terminate with a validation loss of approximately .0003:

```
history = model.fit_generator(train_generator,
steps_per_epoch=num_train_examples // batch_size, epochs=500,
        callbacks=callbacks, validation_data=eval_generator,
        validation_steps=num_eval_examples // batch_size, verbose=2)
```

```
Epoch 1/500
Epoch 00001: val_loss improved from inf to 0.02338, saving model to
model\models\model_model.01-0.0233783.h5
- 442s - loss: 0.0225 - val_loss: 0.0234
Epoch 2/500
Epoch 00002: val_loss improved from 0.02338 to 0.00859, saving model to
model\models\model_model.02-0.0085879.h5
- 37s - loss: 0.0184 - val_loss: 0.0086
Epoch 3/500
Epoch 00003: val_loss improved from 0.00859 to 0.00188, saving model to
model\models\model_model.03-0.0018831.h5
- 38s - loss: 0.0064 - val_loss: 0.0019
……….
```

Our model is now trained and ready to go. Before we see it in action, let's do a quick sanity check and plot some predictions against images:

```
[sample_batch_train_data, sample_batch_test_data] = next(train_generator)
predictions = model.predict([sample_batch_train_data[0],
sample_batch_train_data[1]])
for i in range(0, 3, 1):
    draw_image_with_label(sample_batch_train_data[0][i],
                          sample_batch_test_data[i], predictions[i])
```

We should see an output similar to Figure 16-15. In this figure, the thick line is the predicted output, and the thin line is the label output. Looks like our predictions are fairly accurate (we can also see the actual and predicted values above the images). Time to deploy our model and see it in action.

Figure 16-15. Drawing actual and predicted steering angles on images

Figure 16-15. Drawing actual and predicted steering angles on images

Deploying Our Autonomous Driving Model

All of the steps in this section are also detailed in the Jupyter Notebook *TestModel.ipynb*. Now that we have our model trained, it is time to spin up our simulator and use our model to drive our car around.

As before, begin by importing some libraries and defining paths:

```
from tensorflow.keras.models import load_model
import sys
import numpy as np
import glob
import os

if ('../../PythonClient/' not in sys.path):
    sys.path.insert(0, '../../PythonClient/')
from AirSimClient import *

# << Set this to the path of the model >>
# If None, then the model with the lowest validation loss from training will be
# used
MODEL_PATH = None

if (MODEL_PATH == None):
    models = glob.glob('model/models/*.h5')
    best_model = max(models, key=os.path.getctime)
    MODEL_PATH = best_model

print('Using model {0} for testing.'.format(MODEL_PATH))
```

Next, load the model and connect to AirSim in the Landscape environment. To start the simulator, on a Windows machine, open a PowerShell command window at the location where we unzipped the simulator package and run the following:

```
.\AD_Cookbook_Start_AirSim.ps1 landscape
```

Back in the Jupyter Notebook, run the following to connect the model to the AirSim client. Ensure that the simulator is running *before* kicking this off:

```
model = load_model(MODEL_PATH)
```

```
client = CarClient()
client.confirmConnection()
client.enableApiControl(True)
car_controls = CarControls()
print('Connection established!')
```

With the connection established, let's now set the initial state of the car as well as some buffers used to store the output from the model:

```
car_controls.steering = 0
car_controls.throttle = 0
car_controls.brake = 0

image_buf = np.zeros((1, 59, 255, 3))
state_buf = np.zeros((1,4))
```

The next step is to set up the model to expect an RGB image from the simulator as the input. We need to define a helper function for this:

```
def get_image():
    image_response = client.simGetImages([ImageRequest(0, AirSimImageType.Scene,
False, False)])[0]
    image1d = np.fromstring(image_response.image_data_uint8, dtype=np.uint8)
    image_rgba = image1d.reshape(image_response.height, image_response.width, 4)

    return image_rgba[76:135,0:255,0:3].astype(float)
```

Finally, let's set up an infinite loop for our model to read an image from the simulator along with the current state of the car, predict the steering angle, and send it back to the simulator. Because our model predicts only steering angles, we will need to supply a control signal to maintain speed ourselves. Let's set it up so that the car will attempt to run at a constant 5 m/s:

```
while (True):
    car_state = client.getCarState()

    if (car_state.speed < 5):
        car_controls.throttle = 1.0
    else:
        car_controls.throttle = 0.0

    image_buf[0] = get_image()
    state_buf[0] = np.array([car_controls.steering, car_controls.throttle,
car_controls.brake, car_state.speed])
```

```
    model_output = model.predict([image_buf, state_buf])
    car_controls.steering = round(0.5 * float(model_output[0][0]), 2)

    print('Sending steering = {0}, throttle = {1}'.format(car_controls.steering,
car_controls.throttle))

    client.setCarControls(car_controls)
```

We should see output similar to the following:

```
Sending steering = 0.03, throttle = 1.0
Sending steering = 0.03, throttle = 1.0
Sending steering = 0.03, throttle = 1.0
Sending steering = 0.03, throttle = 1.0
Sending steering = 0.03, throttle = 1.0
Sending steering = -0.1, throttle = 1.0
Sending steering = -0.12, throttle = 1.0
Sending steering = -0.13, throttle = 1.0
Sending steering = -0.13, throttle = 1.0
Sending steering = -0.13, throttle = 1.0
Sending steering = -0.14, throttle = 1.0
Sending steering = -0.15, throttle = 1.0
```

We did it! The car is driving around nicely on the road, keeping to the right side, for the most part, carefully navigating all the sharp turns and instances where it could potentially go off the road. Kudos on training our first-ever autonomous driving model!

Figure 16-16. Trained model driving the car

Before we wrap up, there are a couple of things worth noting. First, notice that the motion of the car is not perfectly smooth. This is because we are working with a regression problem and making a steering angle prediction for every image frame seen by the car. One way to fix this would be to average out predictions over a buffer of consecutive images. Another idea could be to turn this into a classification problem. More specifically, we could define buckets for the steering angles (..., –0.1, –0.05, 0, 0.05, 0.1, ...), bucketize the labels, and

predict the correct bucket for each image.

If we let the model run for a while (a little more than five minutes), we observe that the car eventually veers off the road randomly and crashes. This happens on a portion of the track with a steep incline. Remember our last requirement during the problem setup? Elevation changes require manipulating throttle and brakes. Because our model can control only the steering angle, it doesn’t do too well on steep roads.

predict the correct bucket for each image.

If we let the model run for a while (a little more than five minutes), we observe that the car eventually veers off the road randomly and crashes. This happens on a portion of the track with a steep incline. Remember our last requirement during the problem setup? Elevation changes require manipulating throttle and brakes. Because our model can control only the steering angle, it doesn’t do too well on steep roads.

Further Exploration

Having been trained on the “Hello, World!” scenario, our model is obviously not a perfect driver but don’t be disheartened. Keep in mind that we have barely scratched the surface of the possibilities at the intersection of deep learning and self-driving cars. The fact that we were able to have our car learn to drive around almost perfectly using a very small dataset is something to be proud of!

END-TO-END DEEP LEARNING FOR SELF-DRIVING CARS OF THE REAL WORLD!

At the beginning of 2016, end-to-end deep learning gained popularity among roboticists. The hypothesis was that given enough visual data, a DNN should be able to learn a task in an “end-to-end” manner, mapping inputs directly to actions. Hence, it should be possible to train robots to perform complicated tasks using only image-based input from cameras into DNNs, resulting in actions.

One of the most popular works on this in the context of autonomous driving came from NVIDIA when it introduced its DAVE-2 system to the world. Unlike traditional systems, DAVE-2 used a single DNN to output driving control signals from image pixels (Figure 16-17). NVIDIA demonstrated that a small amount of training data from less than a hundred hours of driving was sufficient to train the car to operate in diverse conditions, on highways and local, residential roads in sunny, cloudy, and rainy conditions. You can find more details on this work at *https://devblogs.nvidia.com/deep-learning-self-driving-cars*.

Figure 16-17. Architecture of NVIDIA's DAVE-2 system (image source)

Here are some new ideas for us to build on top of what we learned in this chapter. You can implement all of these ideas using the setup we already have in place for this chapter.

Figure 16-17. Architecture of NVIDIA's DAVE-2 system (image source)

Here are some new ideas for us to build on top of what we learned in this chapter. You can implement all of these ideas using the setup we already have in place for this chapter.

Expanding Our Dataset

As a general rule, using more data helps improve model performance. Now that we have the simulator up and running, it will be a useful exercise to expand our dataset by doing more data collection runs. We can even try combining data from various different environments available in AirSim and see how our model trained on this data performs in different environments.

We used only RGB data from a single camera in this chapter. AirSim allows us to do a lot more. For example, we can collect images in depth view, segmentation view, surface normal view, and more for each of the cameras available. So, we can potentially have 20 different images (for five cameras operating in all four modes) for each instance. Can using all this extra data help us improve the model we just trained?

Expanding Our Dataset

As a general rule, using more data helps improve model performance. Now that we have the simulator up and running, it will be a useful exercise to expand our dataset by doing more data collection runs. We can even try combining data from various different environments available in AirSim and see how our model trained on this data performs in different environments.

We used only RGB data from a single camera in this chapter. AirSim allows us to do a lot more. For example, we can collect images in depth view, segmentation view, surface normal view, and more for each of the cameras available. So, we can potentially have 20 different images (for five cameras operating in all four modes) for each instance. Can using all this extra data help us improve the model we just trained?

Training on Sequential Data

Our model currently uses a single image and a single vehicle state for each prediction. This is not really how we drive in real life, though. Our actions always take into account the recent series of events leading up to that given moment. In our dataset, we have timestamp information available for all our images, which we can use to create sequences. We can modify our model to make predictions using the previous *N* images and states. For example, given the past 10 images and past 10 states, predict the next steering angle. (Hint: This might require the use of RNNs.)

Training on Sequential Data

Our model currently uses a single image and a single vehicle state for each prediction. This is not really how we drive in real life, though. Our actions always take into account the recent series of events leading up to that given moment. In our dataset, we have timestamp information available for all our images, which we can use to create sequences. We can modify our model to make predictions using the previous N images and states. For example, given the past 10 images and past 10 states, predict the next steering angle. (Hint: This might require the use of RNNs.)

Reinforcement Learning

After we learn about reinforcement learning in the next chapter, we can come back and try the Distributed Deep Reinforcement Learning for Autonomous Driving tutorial from the *Autonomous Driving Cookbook*. Using the Neighborhood environment in AirSim, which is included in the package we downloaded for this chapter, we will learn how to scale a deep reinforcement learning training job and reduce training time from under a week to less than an hour using transfer learning and the cloud.

Reinforcement Learning

After we learn about reinforcement learning in the next chapter, we can come back and try the Distributed Deep Reinforcement Learning for Autonomous Driving tutorial from the *Autonomous Driving Cookbook*. Using the Neighborhood environment in AirSim, which is included in the package we downloaded for this chapter, we will learn how to scale a deep reinforcement learning training job and reduce training time from under a week to less than an hour using transfer learning and the cloud.

Summary

This chapter gave us a sneak peek into how deep learning enables the autonomous driving industry. Taking advantage of the skills acquired in the previous chapters, we implemented the “Hello, World!” problem of autonomous driving using Keras. Exploring the raw data at hand, we learned how to preprocess it to make it suitable for training a high-performing model. And we were able to accomplish this with a very small dataset. We were also able to take our trained model and deploy it to drive a car in the simulated world. Wouldn’t you agree that there’s just something magical about that?

Summary

This chapter gave us a sneak peek into how deep learning enables the autonomous driving industry. Taking advantage of the skills acquired in the previous chapters, we implemented the “Hello, World!” problem of autonomous driving using Keras. Exploring the raw data at hand, we learned how to preprocess it to make it suitable for training a high-performing model. And we were able to accomplish this with a very small dataset. We were also able to take our trained model and deploy it to drive a car in the simulated world. Wouldn’t you agree that there’s just something magical about that?

Chapter 17. Building an Autonomous Car in Under an Hour: Reinforcement Learning with AWS DeepRacer

Contributed by guest author: Sunil Mallya

If you follow technology news, you probably have seen a resurgence in debates about when computers are going to take over the world. Although that's a fun thought exercise, what's triggered the resurgence in these debates? A large part of the credit can be attributed to the news of computers beating humans at decision-making tasks—winning in chess, achieving high scores in video games like Atari (2013), beating humans in a complex game of Go (2016), and, finally, beating human teams at Defence of the Ancients (Dota) 2 in 2017. The most astonishing thing about these successes is that the "bots" learned the games by playing against one another and reinforcing the strategies that they found to bring them success.

If we think more broadly on this concept, it's no different than how humans teach their pets. To train a dog, every good behavior is reinforced by rewarding the dog with a treat and lots of hugs, and every undesired behavior is discouraged by asserting "bad doggie." This concept of reinforcing good behaviors and discouraging bad ones essentially forms the crux of *reinforcement learning*.

Computer games, or games in general, require a sequence of decisions to be made, so traditional supervised methods aren't well suited, because they often focus on making a single decision (e.g., is this an image of a cat or a dog?). The inside joke in the reinforcement learning community is that we just play video games all day (spoiler alert: It's true!). At present, reinforcement learning is being applied across industries to optimize stock trading, manage heating and cooling in large buildings and datacenters, perform real-time bidding of advertisements, optimize video-streaming quality, and even optimize chemical

reactions in a lab. Given these examples of production systems, we would strongly advocate using reinforcement learning for sequential decision making and optimization problems. In this chapter, we focus on learning this paradigm of machine learning and applying it to a real-world problem: building a one-eighteenth-scale, self-driving autonomous car in less than an hour.

–

reactions in a lab. Given these examples of production systems, we would strongly advocate using reinforcement learning for sequential decision making and optimization problems. In this chapter, we focus on learning this paradigm of machine learning and applying it to a real-world problem: building a one-eighteenth-scale, self-driving autonomous car in less than an hour.

A Brief Introduction to Reinforcement Learning

Similar to deep learning, reinforcement learning has seen a renaissance in the past few years ever since previously held human video game records were broken. Reinforcement learning theory had its heyday in the 1990s, but didn't break in to large-scale production systems due to computational requirements and difficulty in training these systems. Traditionally reinforcement learning has been considered compute heavy; in contrast, neural networks are data heavy. But advancements in deep neural networks have also benefited reinforcement learning. Neural networks are now being used to represent the reinforcement learning model, thus giving birth to deep reinforcement learning. In this chapter, we use the term reinforcement learning and deep reinforcement learning interchangeably, but in almost all cases if not stated otherwise, when we refer to reinforcement learning, we are talking about deep reinforcement learning.

Despite recent advancements, the landscape for reinforcement learning has not been developer friendly. Interfaces for training deep learning models have progressively become simpler, but this hasn't quite caught up in the reinforcement learning community. Another challenging aspect for reinforcement learning has been the significant computational requirements and time to model convergence (it has completed learning)—it literally takes days, if not weeks, to create a converged model. Now, assuming that we had the patience, knowledge of neural networks, and the monetary bandwidth, educational resources regarding reinforcement learning are few and far between. Most of the resources are targeted to advanced data scientists, and sometimes out of reach for developers. That one-eighteenth-scale autonomous car we referenced earlier? That's AWS' DeepRacer (Figure 17-1). One of the biggest motivations behind AWS DeepRacer was to make reinforcement learning accessible to developers. It is powered by Amazon SageMaker reinforcement learning, which is a general-purpose reinforcement learning platform. And, let's get real: who doesn't like self-driving cars?

t

Figure 17-1. The AWS DeepRacer one-eighteenth-scale autonomous car

t

Figure 17-1. The AWS DeepRacer one-eighteenth-scale autonomous car

Why Learn Reinforcement Learning with an Autonomous Car?

In recent years, self-driving technology has seen significant investments and success. DIY self-driving and radio-controlled (RC) car-racing communities have since become popular. This unparalleled enthusiasm in developers building scaled autonomous cars with real hardware, testing in real-world scenarios has compelled us to use a “vehicle,” literally, to educate developers on reinforcemen learning. Although there existed other algorithms to build self-driving cars, like traditional computer vision or supervised learning (behavioral cloning), we believe reinforcement learning has an edge over these.

Table 17-1 summarizes some of the popular self-driving kits available for developers, and the technologies that enable them. One of the key benefits of reinforcement learning is that models can be trained exclusively in a simulator. But reinforcement learning systems come with their own set of challenges, the biggest of them being the simulation-to-real (sim2real) problem. There’s always a challenge of deploying models trained completely in simulation in to real-world environments. DeepRacer tackles this with some simple, yet effective solutions that we discuss later in the chapter. Within the first six months following the launch of DeepRacer in November 2018, close to 9,000 developers trained their models in the simulator and successfully tested them on a real-world track.

Table 17-1. Landscape of autonomous self-driving technology

	Hardware	Assembly	Technology	Cost	
AWS DeepRacer	Intel Atom with 100 GFLOPS GPU	Preassembled	Reinforcement learning	$399	
OpenMV	OpenMV H7	DIY (two hours)	Traditional computer vision	$90	

Duckietown	Raspberry Pi	Preassembled	Reinforcement learning, behavioral cloning	$279–$350	
DonkeyCar	Raspberry Pi	DIY (two to three hours)	Behavioral closing	$250	
NVIDIA JetRacer	Jetson Nano	DIY (three to five hours)	Supervised learning	~$400	

FROM THE CREATOR'S DESK

By Chris Anderson, CEO 3DR and founder of DIY Robocars

The future of robocars in DIY communities will be:

Faster
: Consistently beating humans under all conditions, starting with straight speed and getting to track/racing offensive and defensive tactics. We want our cars to be the nonplayer characters of real-world Mario Kart races set to ludicrous difficulty. ;-)

Easier
: In the same way that a great game is "easy to pick up, but hard to master," we want our cars to drive well out of the box but require clever software approaches to win.

Cheaper
: The cost and machine learning barrier to entry in to DIY racing leagues will significantly decrease.

Duckietown	Raspberry Pi	Preassembled	Reinforcement learning, behavioral cloning	$279–$350	
DonkeyCar	Raspberry Pi	DIY (two to three hours)	Behavioral closing	$250	
NVIDIA JetRacer	Jetson Nano	DIY (three to five hours)	Supervised learning	~$400	

FROM THE CREATOR'S DESK

By Chris Anderson, CEO 3DR and founder of DIY Robocars

The future of robocars in DIY communities will be:

Faster

Consistently beating humans under all conditions, starting with straight speed and getting to track/racing offensive and defensive tactics. We want our cars to be the nonplayer characters of real-world Mario Kart races set to ludicrous difficulty. ;-)

Easier

In the same way that a great game is "easy to pick up, but hard to master," we want our cars to drive well out of the box but require clever software approaches to win.

Cheaper

The cost and machine learning barrier to entry in to DIY racing leagues will significantly decrease.

Practical Deep Reinforcement Learning with DeepRacer

Now for the most exciting part of this chapter: building our first reinforcement learning-based autonomous racing model. Before we embark on the journey, let's build a quick cheat sheet of terms that will help you to become familiar with important reinforcement learning terminology:

Goal

Finishing a lap around the track without going off track.

Input

In a human driven car, the human visualizes the environment and uses driving knowledge to make decisions and navigate the road. DeepRacer is also a vision-driven system and so we use a single camera image as our input into the system. Specifically, we use a grayscale 120x160 image as the input.

Output (Actions)

In the real world, we drive the car by using the throttle (gas), brake, and steering wheel. DeepRacer, which is built on top of a RC car, has two control signals: the throttle and the steering, both of which are controlled by traditional pulse-width modulation (PWM) signals. Mapping driving to PWM signals can be nonintuitive, so we discretize the driving actions that the car can take. Remember those old-school car-racing games we played on the computer? The simplest of those used the arrow keys—left, right, and up—to drive the car. Similarly, we can define a fixed set of actions that the car can take, but with more fine-grained control over throttle and steering. Our reinforcement learning model, after it's trained, will make decisions on which action to take such that it can navigate the track successfully. We'll have the flexibility to define these actions as we create the model.

NOTE

A servo in an RC hobby car is generally controlled by a PWM signal, which is a series of pulses of

varying width. The position in which the servo needs to end up is achieved by sending a particular width of the pulse signal. The parameters for the pulses are min pulse width, max pulse width, and repetition rate.

Agent

The system that learns and makes decisions. In our case, it's the car that's learning to navigate the environment (the track).

Environment

Where the agent learns by interacting with actions. In DeepRacer, the environment contains a track that defines where the agent can go and be in. The agent explores the environment to collect data to train the underlying deep reinforcement learning neural network.

State (s)

The representation of where the agent is in an environment. It's a point-in-time snapshot of the agent. For DeepRacer, we use an image as the state.

Actions (a)

Set of decisions that the agent can make.

Step

Discrete transition from one state to the next.

Episode

This refers to an attempt by the car to achieve its goal; that is, complete a lap on the track. Thus, an episode is a sequence of steps, or experience. Differen episodes can have different lengths.

Reward (r)

The value for the action that the agent took given an input state.

Policy (π)

Decision-making strategy or function; a mapping from state to actions.

Value function (V)

The mapping of state to values, in which value represents the expected reward for an action given the state.

Replay or experience buffer

Temporary storage buffer that stores the experience, which is a tuple of (s,a,r,s'), where “s” stands for an observation (or state) captured by the camera, “a” for an action taken by the vehicle, “r” for the expected reward incurred by the said action, and “s'” for the new observation (or new state) after the action is taken.

Reward function

Any reinforcement learning system needs a guide, something that tells the model as it learns what’s a good or a bad action given the situation. The reward function acts as this guide, which evaluates the actions taken by the car and gives it a reward (scalar value) that indicates the desirability of that action for the situation. For example, on a left turn, taking a “left turn” action would be considered optimal (e.g., reward = 1; on a 0–1 scale), but taking a “right turn” action would be bad (reward = 0). The reinforcement learning system eventually collects this guidance based on the reward function and trains the model. This is the most critical piece in training the car and the part that we’ll focus on.

Finally, when we put together the system, the schematic flow looks as follows:

```
Input (120x160 grayscale Image) → (reinforcement learning Model) →
Output (left, right, straight)
```

In AWS DeepRacer, the reward function is an important part of the model building process. We must provide it when training our AWS DeepRacer model.

In an episode, the agent interacts with the track to learn the optimal set of actions it needs to take to maximize the expected cumulative reward. But a single episode doesn’t produce enough data for us to train the agent. So, we end up collecting many episodes’ worth of data. Periodically, at the end of every *n*th episode, we kick off a training process to produce an iteration of the

reinforcement learning model. We run many iterations to produce the best model we can. This process is explained in detail in the next section. After the training, the agent executes autonomous driving by running inference on the model to take an optimal action given an image input. The evaluation of the model can be done in either the simulated environment with a virtual agent or a real-world environment with a physical AWS DeepRacer car.

It's finally time to create our first model. Because the input into the car is fixed, which is a single image from the camera, we need to focus only on the output (actions) and the reward function. We can follow the steps below to begin training the model.

SIMULATION TO REAL TRANSFER PROBLEM (SIM2REAL)

Data collection for robotic applications can be extremely difficult and time consuming. So, there are great advantages to training these applications in a simulated environment. However, creating certain scenarios in the real world can be impractical, like simulating a collision with another car or a human. Therefore, there can be reality gaps in the simulator—perceptual differences and simulation fidelity (image quality, sampling rate, etc.), physical differences (friction, mass, density, etc.), and incorrect physical modeling (interactions and collisions between physical objects in the simulator). All of these can lead to the simulation environment being vastly different from the real world.

reinforcement learning model. We run many iterations to produce the best model we can. This process is explained in detail in the next section. After the training, the agent executes autonomous driving by running inference on the model to take an optimal action given an image input. The evaluation of the model can be done in either the simulated environment with a virtual agent or a real-world environment with a physical AWS DeepRacer car.

It's finally time to create our first model. Because the input into the car is fixed, which is a single image from the camera, we need to focus only on the output (actions) and the reward function. We can follow the steps below to begin training the model.

SIMULATION TO REAL TRANSFER PROBLEM (SIM2REAL)

Data collection for robotic applications can be extremely difficult and time consuming. So, there are great advantages to training these applications in a simulated environment. However, creating certain scenarios in the real world can be impractical, like simulating a collision with another car or a human. Therefore, there can be reality gaps in the simulator—perceptual differences and simulation fidelity (image quality, sampling rate, etc.), physical differences (friction, mass, density, etc.), and incorrect physical modeling (interactions and collisions between physical objects in the simulator). All of these can lead to the simulation environment being vastly different from the real world.

Building Our First Reinforcement Learning

To do this exercise, you will need an AWS account. Log into the AWS Console using your account credentials, as shown in Figure 17-2.

Figure 17-2. The AWS login console

First, let's make sure we are in the North Virginia region, given that the service is available only in that region, and then navigate to the DeepRacer console page: *https://console.aws.amazon.com/deepracer/home?region=us-east-1#getStarted*.

After you select "Reinforcement learning," the model page opens. This page shows a list of all the models that have been created and the status of each model. To create a model, start the process here.

Figure 17-3. Workflow for training the AWS DeepRacer model

Step 1: Create Model

We are going to create a model that can be used by the AWS DeepRacer car to autonomously drive (take actions) around a race track. We need to select the specific race track, provide the actions that our model can choose from, provide a reward function that will be used to incentivize our desired driving behavior, and configure the hyperparameters used during training.

Figure 17-4. Creating a model on the AWS DeepRacer console

Step 1: Create Model

We are going to create a model that can be used by the AWS DeepRacer car to autonomously drive (take actions) around a race track. We need to select the specific race track, provide the actions that our model can choose from, provide a reward function that will be used to incentivize our desired driving behavior, and configure the hyperparameters used during training.

Figure 17-4. Creating a model on the AWS DeepRacer console

Step 2: Configure Training

In this step, we select our training environment, configure action space, write ou reward function, and adjust other training related settings before kicking off our training job.

Configure the simulation environment

Training our reinforcement learning model takes place on a simulated race track, and we can choose the track to train our model. We'll use AWS RoboMaker, a cloud service that makes building robotic applications easy, to spin up the simulation environment.

When training a model, we pick the track most similar to the final track we intend to race on. As of July 2019, AWS DeepRacer provides seven tracks that we can train on. While configuring such a complementary environment isn't required and doesn't guarantee a good model, it will maximize the odds that our model will perform best on the race track. Furthermore, if we train on a straight track, it's unlikely that our model would have learned how to turn. Just like in the case of supervised learning in which it's unlikely that the model will learn something that's not part of the training data, in reinforcement learning, the agent is unlikely to learn anything out of scope from the training environment. For our first exercise, select the re:Invent 2018 track, as shown in Figure 17-5.

To train a reinforcement learning model, we must choose a learning algorithm. Currently, the AWS DeepRacer console supports only the proximal policy optimization (PPO) algorithm. The team eventually will support more algorithms, but PPO was chosen for faster training times and superior convergence properties. Training a reinforcement learning model is an iterative process. First, it's a challenge to define a reward function to cover all important behaviors of an agent in an environment at once. Second, hyperparameters are often tuned to ensure satisfactory training performance. Both require experimentation. A prudent approach is to start with a simple reward function, which will be our approach in this chapter, and then progressively enhance it. AWS DeepRacer facilitates this iterative process by enabling us to clone a trained model, in which we could enhance the reward function to handle previous ignored variables or we can systematically adjust hyperparameters unti

the result converges. The easiest way to detect this convergence is look at the logs and see whether the car is going past the finish line; in other words, whether progress is 100%. Alternatively, we could visually observe the car's behavior and confirm that it goes past the finish line.

Figure 17-5. Track selection on the AWS DeepRacer console

Configure the action space

Next, we configure the action space that our model will select both during and after training. An action (output) is a combination of speed and steering angle. Currently in AWS DeepRacer, we use a discrete action space (a fixed set of actions) as opposed to a continuous action space (turn *x* degrees with *y* speed, where *x* and *y* take real values). This is because it's easier to map to values on the physical car, which we dive into later in the "Racing the AWS DeepRacer

Car” To build this discrete action space, we specify the maximum speed, the speed levels, the maximum steering angle, and the steering levels, as depicted in Figure 17-6.

aws Services Resource Groups N. Virginia

Action space Info

Action space defines the specifc actions an agent can take in both the simulator and physical world. While a real vehicle can choose from a continuum of actions, AWS DeepRacer simplifies the agent's decision-making process by reducing that space to a set of discrete actions.

Configure this discrete action space by setting the range and granularity for speed and steering angle. The system automatically generates an action space according to that specification. Note that your model will take longer to train under a larger action space.

Maximum steering angle

30 degrees

Max values are between 1 and 30.

Steering angle granularity

5

Maximum speed

3 m/s

Max values are between 0.8 and 8.

Speed granularity

3

Action list

Action number	Steering	Speed
0	-30 degrees	1 m/s
1	-30 degrees	2 m/s
2	-30 degrees	3 m/s
3	-15 degrees	1 m/s
4	-15 degrees	2 m/s
5	-15 degrees	3 m/s
6	0 degrees	1 m/s
7	0 degrees	2 m/s
8	0 degrees	3 m/s
9	15 degrees	1 m/s
10	15 degrees	2 m/s
11	15 degrees	3 m/s
12	30 degrees	1 m/s
13	30 degrees	2 m/s
14	30 degrees	3 m/s

Figure 17-6. Defining the action space on the AWS DeepRacer console

Following are the configuration parameters for the action space:

Maximum steering angle

This is the maximum angle in degrees that the front wheels of the car can turn to the left and to the right. There is a limit as to how far the wheels can turn, and so the maximum turning angle is 30 degrees.

Steering angle granularity

Refers to the number of steering intervals between the maximum steering angle on either side. Thus, if our maximum steering angle is 30 degrees, +30 degrees is to the left and –30 degrees is to the right. With a steering granularity of 5, the following steering angles, as shown in Figure 17-6, from left to right, will be in the action space: 30 degrees, 15 degrees, 0 degrees, –15 degrees, and –30 degrees. Steering angles are always symmetrical around 0 degrees.

Maximum speed

Refers to the maximum speed the car will drive in the simulator as measured in meters per second (m/s).

Speed levels

Refers to the number of speed levels from the maximum speed (including) to zero (excluding). So, if our maximum speed is 3 m/s and our speed granularity is 3, our action space will contain speed settings of 1 m/s, 2 m/s, and 3 m/s. Simply put, 3 m/s divided by 3 = 1 m/s, so go from 0 m/s to 3 m/s in increments of 1 m/s. 0 m/s is not included in the action space.

Based on the previous example the final action space will include 15 discrete actions (three speeds x five steering angles), which should be listed in the AWS DeepRacer service. Feel free to tinker with other options, just remember that larger action spaces might take a bit longer to train.

TIP

Based on our experience, here are some tips on how to configure the action space:

- Our experiments have shown that models with a faster maximum speed take longer to converge than those with a slower maximum speed. In some cases (reward function and track dependent), it can take longer than 12 hours for a 5 m/s model to converge.
- Our model will not perform an action that is not in the action space. Similarly, if the model is trained on a track that never required the use of this action—for example, turning won't be incentivized on a straight track—the model won't know how to use this action, because it won't be incentivized to turn. As you begin thinking about building a robust model, make sure that you keep the action space and training track in mind.
- Specifying a fast speed or a wide steering angle is great, but we still need to think about our reward function and whether it makes sense to drive full speed into a turn, or exhibit zigzag behavior on a straight section of the track.
- We also need to keep physics in mind. If we try to train a model at faster than 5 m/s, we might see our car spin out on corners, which will probably increase the time to convergence of our model.

Configure reward function

As we explained earlier, the reward function evaluates the quality of an action's outcome given the situation, and rewards the action accordingly. In practice the reward is calculated during training after each action is taken, and forms a key part of the experience used to train the model. We then store the tuple (state, action, next state, reward) in a memory buffer. We can build the reward function logic using a number of variables that are exposed by the simulator. These variables represent measurements of the car, such as steering angle and speed; the car in relation to the racetrack, such as (x, y) coordinates; and the racetrack, such as waypoints (milestone markers on the track). We can use these measurements to build our reward function logic in Python 3 syntax.

All of the parameters are available as a dictionary to the reward function. Their keys, data types, and descriptions are documented in Figure 17-7, and some of the more nuanced ones are further illustrated in Figure 17-8.

```
{
    "all_wheels_on_track": Boolean,    # flag to indicate if the vehicle is on the track
    "x": float,                        # vehicle's x-coordinate in meters
    "y": float,                        # vehicle's y-coordinate in meters
    "distance_from_center": float,     # distance in meters from the track center
    "is_left_of_center": Boolean,      # Flag to indicate if the vehicle is on the left side to the track center or not.
    "heading": float,                  # vehicle's yaw in degrees
    "progress": float,                 # percentage of track completed
    "steps": int,                      # number steps completed
    "speed": float,                    # vehicle's speed in meters per second (m/s)
    "steering_angle": float,           # vehicle's steering angle in degrees
    "track_width": float,              # width of the track
    "waypoints": [[float, float], … ], # list of [x,y] as milestones along the track center
    "closest_waypoints": [int, int]    # indices of the two nearest waypoints.
}
```

Figure 17-7. Reward function parameters (a more in-depth review of these parameters is available in the

documentation)

Figure 17-8. Visual explanation of some of the reward function parameters

To build our first model, let's pick an example reward function and train our model. Let's use the default template, shown in Figure 17-9, in which the car tries to follow the center dashed lines. The intuition behind this reward function is to take the safest navigation path around the track because being in the center keeps the car farthest from being off track. The reward function does the following: creates three tiers around the track, using the three markers, and then proceeds to reward the car more for driving in the second tier as opposed to the center or the last tier. Also note the differences in the size of the reward. We provide a reward of 1 for staying in the narrow center tier, 0.5 for staying in the second (off-center) tier, and 0.1 for staying in the last tier. If we decrease the reward for the center tier, or increase the reward for the second tier, we are essentially incentivizing the car to use a larger portion of the track surface. Remember that time you did your driving test? The examiner probably did the same and cut off points as you got closer to the curb or the lane makers. This could come in handy, especially when there are sharp corners.

Figure 17-9. An example reward function

Here's the code that sets this up:

```
def reward_function(params):
    '''
    Example of rewarding the agent to follow center line
    '''

    # Read input parameters
    track_width = params['track_width']
    distance_from_center = params['distance_from_center']

    # Calculate 3 markers that are at varying distances away from the center line
    marker_1 = 0.1 * track_width
    marker_2 = 0.25 * track_width
    marker_3 = 0.5 * track_width

    # Give higher reward if the car is closer to center line and vice versa
    if distance_from_center <= marker_1:
        reward = 1.0
    elif distance_from_center <= marker_2:
        reward = 0.5
    elif distance_from_center <= marker_3:
        reward = 0.1
    else:
        reward = 1e-3 # likely crashed/ close to off track

    return float(reward)
```

Because this is the first training run, let's focus on understanding the process of creating and evaluating a basic model, and then focus on optimizing it. In this case, we skip the algorithm settings and hyperparameter sections and use the defaults.

NOTE

FAQ: Should the rewards be in a certain range, and also can I give negative rewards?

There are no real constraints on what we can reward and not reward, but as a good practice it's easier to understand rewards when they are in a 0–1 or 0–100 scale. What's more important is that our reward scale gives relative rewards for the actions appropriately. For example, on a right turn, we should reward the right action with high reward, a left action with close to 0 reward, perhaps a straight action with an in-between reward or higher than the left action because it might not be a completely bad action to take.

Configure stop conditions

This is the last section before we begin training. Here we specify the maximum time our model will train for. This is provided as a convenient mechanism for us to terminate our training given that we are billed for the amount of training time.

Specify 60 minutes and then select "Start training." If there is an error, we will be taken to the error location. After we start training, it can take up to six minutes to spin up the services (such as Amazon SageMaker, AWS Robomaker, AWS Lambda, AWS Step Function) needed to start training. Remember, we can always stop training early if we determine that the model has converged (as explained in the next section) by clicking the "stop" button.

Because this is the first training run, let's focus on understanding the process of creating and evaluating a basic model, and then focus on optimizing it. In this case, we skip the algorithm settings and hyperparameter sections and use the defaults.

NOTE

FAQ: Should the rewards be in a certain range, and also can I give negative rewards?

There are no real constraints on what we can reward and not reward, but as a good practice it's easier to understand rewards when they are in a 0–1 or 0–100 scale. What's more important is that our reward scale gives relative rewards for the actions appropriately. For example, on a right turn, we should reward the right action with high reward, a left action with close to 0 reward, perhaps a straight action with an in-between reward or higher than the left action because it might not be a completely bad action to take.

Configure stop conditions

This is the last section before we begin training. Here we specify the maximum time our model will train for. This is provided as a convenient mechanism for us to terminate our training given that we are billed for the amount of training time.

Specify 60 minutes and then select "Start training." If there is an error, we will be taken to the error location. After we start training, it can take up to six minutes to spin up the services (such as Amazon SageMaker, AWS Robomaker, AWS Lambda, AWS Step Function) needed to start training. Remember, we can always stop training early if we determine that the model has converged (as explained in the next section) by clicking the "stop" button.

Step 3: Model Training

After our model has begun training, we can select it from the listed models on the DeepRacer console. We can then see quantitatively how the training is progressing by looking at the total reward over time graph and also qualitatively from a first-person view from the car in the simulator (see Figure 17-10).

At first, our car will not be able to drive on a straight road, but as it learns better driving behavior, we should see its performance improving and the reward graph increasing. Furthermore, when our car drives off of the track it will be reset on the track. We might observe that the reward graph is spiky.

> **NOTE**
>
> *FAQ: Why is the reward graph spiky?*
>
> The agent starts with high exploration, and gradually begins to exploit the trained model. Because the agent always takes random actions for a fraction of its decisions, there might be occasions for which it totally makes the wrong decision and ends up going off track. This is typically high at the beginning of training, but eventually the spikiness should reduce as the model begins to learn.

Logs are always a good source of more granular information regarding our model's training. Later in the chapter, we examine how we can use the logs programmatically to gain a deeper understanding of our model training. In the meantime, we can look at the log files for both Amazon SageMaker and AWS RoboMaker. The logs are output to Amazon CloudWatch. To see the logs, hover your mouse over the reward graph and select the three dots that appear below the refresh button. Then, select "View logs." Because the model training will take an hour, this might be a good time to skip to the next section and learn a bit more about reinforcement learning.

Figure 17-10. Training graph and simulation video stream on the AWS DeepRacer console

Figure 17-10. Training graph and simulation video stream on the AWS DeepRacer console

Step 4: Evaluating the Performance of the Model

In reinforcement learning, the best way to gauge the ability of the model is to run it such that it only exploits—that is, it doesn't take random actions. In our case, first test it on a track similar to the one it's trained on to see whether it can replicate the trained behavior. Next, try it on a different track to test generalizability. After our model training is complete, we can commence model evaluation. From our model details page, where we observed training, select "Start evaluation." We can now select the track on which we want to evaluate the performance of our model and also the number of laps. Select the "re:Invent 2018" track and 5 laps and then select Start. When it's done, we should see a similar page as that shown in Figure 17-11, which summarizes the results of our model's attempts to go around the track and complete the lap.

Figure 17-11. Model evaluation page on the AWS DeepRacer console

Great job! We have successfully built our first reinforcement learning–enabled autonomous car.

> **NOTE**
>
> *FAQ: When I run evaluation, I see only 100% completion rates sometimes?*
>
> As the model runs inference and navigates around the track in the simulator, due to the fidelity of the simulator, the car can end up in slightly different positions for the same action—in other words, a 15 degree left turn action might result in only 14.9 degrees. Practically, we observe only very small deviations in the simulator, but these small deviations can add up over time. A well-trained model is able to recover from close-to-off-track positions, but an undertrained model is likely to not recover from close calls.

Now that our model has been successfully evaluated, we move on to improving it and learn to achieve better lap times. But before that, it's necessary for you to understand more about the theory behind reinforcement learning and dig deeper into the learning process.

NOTE

We can get started with the AWS DeepRacer console service to create our first model, train it (for up to six hours), evaluate it, and submit it to the AWS DeepRacer League for free.

Now that our model has been successfully evaluated, we move on to improving it and learn to achieve better lap times. But before that, it's necessary for you to understand more about the theory behind reinforcement learning and dig deeper into the learning process.

NOTE

We can get started with the AWS DeepRacer console service to create our first model, train it (for up to six hours), evaluate it, and submit it to the AWS DeepRacer League for free.

Reinforcement Learning in Action

Let’s take a closer look at reinforcement learning in action. Here, we discuss some theory, the inner workings, and lots of practical insights related to our autonomous car project.

Reinforcement Learning in Action

Let’s take a closer look at reinforcement learning in action. Here, we discuss some theory, the inner workings, and lots of practical insights related to our autonomous car project.

How Does a Reinforcement Learning System Learn?

First, we must understand *explore* versus *exploit*. Similar to how a child learns, a reinforcement learning system learns from *exploring* and finding out what's good and bad. In the case of the child, the parent guides or informs the child of its mistakes, qualifies the decisions made as good or bad, or to what degree they were good or bad. The child memorizes the decisions it took for given situations and tries to replay, or *exploit*, those decisions at opportune moments. In essence, the child is trying to maximize the parent's approval or cheer. Early in a child's life, it's more attentive to the suggestions from its parents, thus creating opportunities to learn. And later in life, as adults who seldom listen to their parents, exploit the concepts they have learned to make decisions.

As shown in Figure 17-12, at every timestep "t," the DeepRacer car (agent) receives its current observation, the state (S_t), and on that basis the car selects an action (A_t). As a result of the action the agent took, it gets a reward R_{t+1} and moves to state S_{t+1}, and this process continues throughout the episode.

Figure 17-12. Reinforcement learning theory basics in a nutshell

In the context of DeepRacer, the agent explores by taking random actions, and the reward function, which essentially acts just like the parent, tells the car whether the actions it took for the given state were good or not. Usually, "goodness" of the action for a given state is expressed as a number, with higher numbers meaning it's close to optimal and lower meaning it's not. The system records all of this information for every step, specifically: *current state, action taken, reward for the action, and next state* (s,a,r,s'), in what we call the experience replay buffer, which is essentially a temporary memory buffer. The entire idea here being that the car can learn from what decisions were good and

other decisions that were bad. The key point is that we *start with a high degree of exploration, and slowly increase exploitation.*

In the DeepRacer simulator, we sample the input image state at 15 frames per second. At each step or image frame captured, the car transitions from one state to another. Each image represents the state the car is in, and eventually after the reinforcement learning model is trained, it will look to exploit by inferring which action to take. To make decisions, we could either take random actions or use our model for a recommendation. As the model trains, the training process balances the exploration and exploitation. To begin, we explore more, given that our model is unlikely to be good, but as it learns through experience, we shift more toward exploitation by giving the model more control. Figure 17-13 depicts this flow. This transition could be a linear decay, an exponential decay, or any similar strategy that's usually tuned based on the degree of learning we hypothesize. Typically, an exponential decay is used in practical reinforcement learning training.

Figure 17-13. The DeepRacer training flow

After an action is taken by a random choice or the model prediction, the car transitions to a new state. Using the reward function, the reward is computed and assigned to the outcome. This process will continue, for each state, until we get to a terminal state; that is, the car goes off track or completes a lap, at which point the car will be reset and then repeated. A step is a transition from one state to another, and at each step a (state, action, new state, reward) tuple is recorded. The collection of steps from the reset point until the terminal state is called an *episode*.

To illustrate an episode, let's look at the example of a miniature race track in Figure 17-14, where the reward function incentivizes following the centerline

because it’s the shortest and quickest path from start to finish. This episode consists of four steps: at step 1 and 2 the car follows the center line, and then it turns left 45 degrees at step 3, and continues in that direction only to finally crash at step 4.

Figure 17-14. Illustration of an agent exploring during an episode

We can think of these episodes as experience, or training data for our model. Periodically, the reinforcement learning system takes a random subset of these recordings from the memory buffer and trains a DNN (our deep reinforcement learning model) to produce a model that can best navigate the environment based on our reward function “guide”; in other words, get maximum cumulative rewards. As time progresses or with more episodes, we see a model that becomes better over time. Of course, the supercritical caveat being that the reward function is well defined and directing the agent toward the goal. If our reward function is bad, our model will not learn the correct behavior.

Let’s take a slight diversion to understand a bad reward function. As we were initially designing the system, the car had freedom to go in any direction (left, right, forward, back). Our simplest reward function didn’t distinguish between directions, and in the end the model learned the best way to accumulate awards by just rocking backward and forward. This was then overcome by incentivizing the car to go forward. Luckily for developers now, the DeepRacer team made it simpler by allowing the car to move only in the forward direction, so we don’t

even need to think of such behavior.

Now back to reinforcement learning. *How do we know if the model is getting better?* Let's return to the concepts of explore and exploit. Remember that we begin with high exploration, low exploitation, and gradually increase the rate of exploitation. This means that at any point in the training process, for a certain percentage of time the reinforcement learning system will explore; that is, take random actions. As the experience buffer grows, for any given state, it stores all kinds of decisions (actions) including the ones that resulted in the highest reward to the lowest reward. The model essentially uses this information to learn and predict the best action to take for the given state. Suppose that the model initially took an action "go straight" for state S and received a reward of 0.5, but the next time it arrived in state S, it took action "go right" and received a reward of 1. If the memory buffer had several samples of the reward as "1" for the same state S for action "go right," the model eventually learns "go right" is the optimal action for that state. Over time, the model will explore less and exploit more, and this percentage allocation is changed either linearly or exponentially to allow for the best learning.

The key here is that if the model is taking optimal actions, the system learns to keep selecting those actions; if it isn't selecting the best action, the system continues to try to learn the best action for a given state. Practically speaking, there can exist many paths to get to the goal, but for the mini racetrack example in Figure 17-13, the fastest path is going to be right down the middle of the track, as practically speaking any turns would slow down the car. During training, in early episodes as the model begins to learn, we observe that it might end up at the finish line (goal) but might not be taking the optimal path, as illustrated in Figure 17-15 (left) where the car zigzags, thus taking more time to get to the finish line and earning cumulative rewards of only 9. But because the system still continues to explore for a fraction of the time, the agent gives itself opportunities to find better paths. As more experience is built, the agent learns to find an optimal path and converge to an optimal policy to reach the goal with total cumulative rewards of 18, as illustrated in Figure 17-15 (right).

0.2
2
0.2
0.2
0.

0.2
2
0.2
0.2
0.

Reinforcement Learning Theory

Now that we understand how a reinforcement learning system learns and works, especially in the context of AWS DeepRacer, let's look at some formal definitions and general reinforcement learning theory. This background will be handy when we solve other problems using reinforcement learning.

The Markov decision process

Markov decision process (MDP) is a discrete stochastic state-transition process framework that is used for modeling decision making in a control process. The markov property defines that each state is solely dependent on the previous state This is a convenient property because it means that to make a state transition, all of the necessary information must be available in the current state. Theoretical results in reinforcement learning rely on the problem being formulated as an MDP, hence it's important to understand how to model a problem as an MDP to be solved using reinforcement learning.

Model free versus model based

Model, in this context, refers to the learned representation of the environment. This is useful because we can potentially learn the dynamics in the environment and train our agent using the model rather than having to use the real environment every time. However, in reality, learning the environment isn't easy and it's often easier to have a representation of the real world in simulation and then jointly learn both perception and dynamics as part of the agent navigation rather than one after the other in a sequence. In this chapter, we focus only on model-free reinforcement learning.

Value based

For every action taken by the agent, there's a corresponding reward assigned by the reward function. For any given state-action pair, it's helpful to know its value (reward). If such a function were to exist, we could compute the maximum reward that can be achieved in any state and simply select the corresponding action to navigate the environment. For example, in a game of 3x3 tic-tac-toe, there are a finite number of game situations, so we could build a lookup table to

give us the best move given the situation. But in a game of chess, given the size of the board and complexity of the game, such a lookup table would be computationally expensive to build and memory storage would be large. Thus, in a complex environment it's difficult to tabulate state-action value pairs or define a function that can map state-action pairs to values. So, instead we try to use a neural network by parameterizing the value function and using a neural network to approximate the value for every action given a state observation. An example of a value-based algorithm is Deep Q-Learning.

Policy based

A policy is a set of rules that the agent learns to navigate the environment. Simply put the policy function tells the agent which action is the best action to take from its current state. Policy-based reinforcement learning algorithms like REINFORCE and Policy Gradients find the optimal policy without the need to map values to states. In reinforcement learning, we parameterize the policy. In other words, we allow a neural network to learn what the optimal policy function is.

Policy based or value based—why not both?

There's always been a debate about whether to use policy-based or value-based reinforcement learning. Newer architectures try to learn both the value function and policy function together rather than keeping one fixed. This approach in reinforcement learning is called *actor critic*.

You can associate the actor with the policy and the critic with the value function. The actor is responsible for taking actions, and the critic is responsible for estimating the "goodness," or the value of those actions. The actor maps states to actions, and the critic maps state-action pairs to values. In the actor-critic paradigm, the two networks (actor and critic) are trained separately using gradient ascent to update the weights of our deep neural network. (Remember that our goal is to maximize cumulative rewards; thus, we need to find global maxima.) As the episodes roll by, the actor becomes better at taking actions that lead to higher reward states, and the critic also becomes better at estimating the values of those actions. The signal for both the actor and critic to learn comes purely from the reward function.

The value for a given state-action pair is called *Q value*, denoted by Q(s,a). We can decompose the Q value into two parts: the estimated value and a quantitative measure of the factor by which the action is better than others. This measure is called the *Advantage* function. We can view Advantage as the difference between the actual reward for a given state-action pair and the expected reward at that state. The higher the difference, the farther we are from selecting an optimal action.

Given that estimating the value of the state could end up being a difficult problem, we can focus on learning the advantage function. This allows us to evaluate the action not only based on how good it is, but also based on how much better it could be. This allows us to converge to an optimal policy much easier than other simpler policy gradient-based methods because, generally, policy networks have high variance.

Delayed rewards and discount factor (γ)

Rewards are given for every state transition based on the action taken. But the impact of these rewards might be nonlinear. For certain problems the immediate rewards might be more important, and in some cases the future rewards more important. As an example, if we were to build a stock-trading algorithm, the future rewards might have higher uncertainty; thus, we would need to discount appropriately. The discount factor (γ) is a multiplication factor between [0,1], where closer to zero indicates rewards in the immediate future are more important. With a high value that’s closer to 1, the agent will focus on taking actions that maximize future rewards.

The value for a given state-action pair is called *Q value*, denoted by Q(s,a). We can decompose the Q value into two parts: the estimated value and a quantitative measure of the factor by which the action is better than others. This measure is called the *Advantage* function. We can view Advantage as the difference between the actual reward for a given state-action pair and the expected reward at that state. The higher the difference, the farther we are from selecting an optimal action.

Given that estimating the value of the state could end up being a difficult problem, we can focus on learning the advantage function. This allows us to evaluate the action not only based on how good it is, but also based on how much better it could be. This allows us to converge to an optimal policy much easier than other simpler policy gradient-based methods because, generally, policy networks have high variance.

Delayed rewards and discount factor (γ)

Rewards are given for every state transition based on the action taken. But the impact of these rewards might be nonlinear. For certain problems the immediate rewards might be more important, and in some cases the future rewards more important. As an example, if we were to build a stock-trading algorithm, the future rewards might have higher uncertainty; thus, we would need to discount appropriately. The discount factor (γ) is a multiplication factor between [0,1], where closer to zero indicates rewards in the immediate future are more important. With a high value that’s closer to 1, the agent will focus on taking actions that maximize future rewards.

Reinforcement Learning Algorithm in AWS DeepRacer

First, let's look at one of the simplest examples of a policy optimization reinforcement learning algorithm: vanilla policy gradient.

Figure 17-16. Training process for the vanilla policy gradient algorithm

We can think of a deep reinforcement learning model as consisting of two parts: the input embedder and policy network. The input embedder will extract features from the image input and pass it to the policy network, which makes decisions; for instance, predict which action is the best for the given input state. Given that our input is an image, we use convolution layers (CNNs) to extract features. Because the policy is something we want to learn, we parameterize our policy function, the simplest of which can be learned with a fully connected layer. The input CNN layers take in an image and then the policy network uses the image features as input and outputs an action. Thus, mapping state to action. As the model trains, we become better at mapping the input space and extracting relevant features and also optimizing the policy to get the best action for each state. Our goal here is to collect maximum cumulative rewards. To achieve that, our model weights are updated such to maximize the cumulative future reward, and in doing so, we give a higher probability to the action that leads to the highe cumulative future reward. In training neural networks previously, we used stochastic gradient descent or its variation; when training an reinforcement learning system, we seek to maximize the cumulative reward; so, instead of minimization, we look to maximize. So, we use gradient ascent to move the

weights in the direction of the steepest reward signal.

DeepRacer uses an advanced variation of policy optimization, called Proximal Policy Optimization (PPO), summarized in Figure 17-17.

Figure 17-17. Training using the PPO algorithm

On the left side of Figure 17-17, we have our simulator using the latest policy (model) to get new experience (s,a,r,s'). The experience is fed into an experience replay buffer that feeds our PPO algorithm after we have a set number of episodes.

On the right side of Figure 17-17, we update our model using PPO. Although PPO is a policy optimization method, it uses the advantage actor-critic method, which we described earlier. We compute the PPO gradient and shift policy in the direction where we get the highest rewards. Blindly taking large steps in this direction can cause too much variability in training; if we take small steps training can take forever. PPO improves the stability of the policy (actor) by limiting how much the policy can update at each training step. This is done by using a clipped surrogate objective function, which prevents the policy from being updated too much, thus solving large variance, a common issue with policy optimization methods. Typically, for PPO, we keep the ratio of new and old policy at [0.8, 1.2]. The critic tells the actor how good the action taken was, and how the actor should adjust its network. After the policy is updated the new model is sent to the simulator to get more experience.

Deep Reinforcement Learning Summary with DeepRacer as an Example

To solve any problem with reinforcement learning, we need to work through the following steps:

1. Define the goal.
2. Select the input state.
3. Define the action space.
4. Construct the reward function.
5. Define the DNN architecture.
6. Pick the reinforcement learning optimization algorithm (DQN, PPO, etc.).

The fundamental manner in which a reinforcement learning model trains doesn't change if we are building a self-driving car or building a robotic arm to grasp objects. This is a huge benefit of the paradigm because it allows us to focus on a much higher level of abstraction. To solve a problem using reinforcement learning, the main task at hand is to define our problem as an MDP, followed by defining the input state and set of actions that the agent can take in a given environment, and the reward function. Practically speaking, the reward function can be one of the most difficult parts to define, and often is the most important a this is what influences the policy that our agent learns. After all the environment dependent factors are defined, we can then focus on what the deep neural network architecture should be to map the input to actions, followed by picking the reinforcement learning algorithm (value-based, policy-based, actor-critic-based) to learn. After we pick the algorithm, we can focus on the high level knobs to control what the algorithm does. When we drive a car, we tend to focus on controls, and the understanding of the internal combustion engine doesn't influence too much the way we drive. Similarly, as long as we understand the knobs that each algorithm exposes, we can train a reinforcement learning model.

It's time to bring this home. Let's formulate the DeepRacer racing problem:

1. Goal: To finish a lap by going around the track in the least amount of time

2. Input: Grayscale 120x160 image
3. Actions: Discrete actions with combined speed and steering angle values
4. Rewards: Reward the car for being on the track, incentivize going faster, and prevent from doing a lot of corrections or zigzag behavior
5. DNN architecture: Three-layer CNN + fully connected layer (Input → CNN → CNN → CNN → FC → Output)
6. Optimization algorithm: PPO

2. Input: Grayscale 120x160 image
3. Actions: Discrete actions with combined speed and steering angle values
4. Rewards: Reward the car for being on the track, incentivize going faster, and prevent from doing a lot of corrections or zigzag behavior
5. DNN architecture: Three-layer CNN + fully connected layer (Input → CNN → CNN → CNN → FC → Output)
6. Optimization algorithm: PPO

Step 5: Improving Reinforcement Learning Models

We can now look to improve our models and also get insights into our model training. First, we focus on training improvements in the console. We have at our disposal the ability to change the reinforcement learning algorithm settings and neural network hyperparameters.

Algorithm settings

This section specifies the hyperparameters that will be used by the reinforcement learning algorithm during training. Hyperparameters are used to improve training performance.

Hyperparameters for the neural network

Table 17-2 presents the hyperparameters that are available to tune the neural network. Even though the default values have experimentally proven to be good, practically speaking the developer should focus on the batch size, number of epochs, and the learning rate as they were found to be the most influential in producing high-quality models; that is, getting the best out of our reward function.

Table 17-2. Description and guidance for tuneable hyperparameters for the deep neural network

Parameter	Description	Tips
Batch size	The number recent of vehicle experiences sampled at random from an experience buffer and used for updating the underlying deep learning neural network weights. If we have 5,120 experiences in the buffer, and specify a batch size of 512, then ignoring random sampling, we will get 10 batches of experience. Each batch will be used, in turn, to update our neural network weights during training.	Use a larger batch size to promote more stable and smooth updates to the neural network weights, but be aware of the possibility that the training may be slower.
Number of epochs	An epoch represents one pass through all batches, where the neural network weights are updated after each batch is processed, before proceeding to the next batch. Ten epochs implies we update the neural network weights, using all batches one at a time, but repeat this process 10 times.	Use a larger number of epochs to promote more stable updates, but expect slower training. When the batch size is small, we can use a smaller number of epochs.

Learning rate	The learning rate controls how big the updates to the neural network weights are. Simply put, when we need to change the weights of our policy to get to the maximum cumulative reward, how much should we shift our policy.	A larger learning rate will lead to faster training, but it may struggle to converge. Smaller learning rates lead to stable convergence, but can take a long time to train.
Exploration	This refers to the method used to determine the trade-off between exploration and exploitation. In other words, what method should we use to determine when we should stop exploring (randomly choosing actions) and when should we exploit the experience we have built up.	Since we will be using a discrete action space, we should always select "CategoricalParameters."
Entropy	A degree of uncertainty, or randomness, added to the probability distribution of the action space. This helps promote the selection of random actions to explore the state/action space more broadly.	
Discount factor	A factor that specifies how much the future rewards contribute to the expected cumulative reward. The larger the discount factor, the farther out the model looks to determine the expected cumulative reward and the slower the training. With a discount factor of 0.9, the vehicle includes rewards from an order of 10 future steps to make a move. With a discount factor of 0.999, the vehicle considers rewards from an order of 1,000 future steps to make a move.	The recommended discount factor values are 0.99, 0.999, and 0.9999.
Loss type	The loss type specifies the type of the objective function (cost function) used to update the network weights. The Huber and mean squared error loss types behave similarly for small updates. But as the updates become larger, the Huber loss takes smaller increments compared to the mean squared error loss.	When we have convergence problems, use the Huber loss type. When convergence is good and we want to train faster, use the mean squared error loss type.
Number of episodes between each training	This parameter controls how much experience the car should obtain between each model training iteration. For more complex problems that have more local maxima, a larger experience buffer is necessary to provide more uncorrelated data points. In this case, training will be slower but more stable.	The recommended values are 10, 20, and 40.

Insights into model training

After the model is trained, at a macro level the rewards over time graph, like the one in Figure 17-10, gives us an idea of how the training progressed and the point at which the model begins to converge. But it doesn't give us an indication

of the converged policy, insights into how our reward function behaved, or areas where the speed of the car could be improved. To help gain more insights, we developed a Jupyter Notebook that analyzes the training logs, and provides suggestions. In this section, we look at some of the more useful visualization tools that can be used to gain insight into our model's training.

The log file records every step that the car takes. At each step it records the x,y location of the car, yaw (rotation), the steering angle, throttle, the progress from the start line, action taken, reward, the closest waypoint, and so on.

Heatmap visualization

For complex reward functions, we might want to understand reward distribution on the track; that is, where did the reward function give rewards to the car on the track, and its magnitude. To visualize this, we can generate a heatmap, as shown in Figure 17-18. Given that the reward function we used gave the maximum rewards closest to the center of the track, we see that region to be bright and a tiny band on either side of the centerline beyond it to be red, indicating fewer rewards. Finally, the rest of the track is dark, indicating no rewards or rewards close to 0 were given when the car was in those locations. We can follow the code to generate our own heatmap and investigate our reward distribution.

Figure 17-18. Heatmap visualization for the example centerline reward function

Improving the speed of our model

After we run an evaluation, we get the result with lap times. At this point, we might be curious as to the path taken by the car or where it failed, or perhaps the

points where it slowed down. To understand all of these, we can use the code in this notebook to plot a race heatmap. In the example that follows, we can observe the path that the car took to navigate around the track and also visualize the speeds at which it passes through at various points on the track. This gives us insight into parts that we can optimize. One quick look at Figure 17-19 (left) indicates that the car didn't really go fast at the straight part of the track; this provides us with an opportunity. We could incentivize the model by giving more rewards to go faster at this part of the track.

Figure 17-19. Speed heatmap of an evaluation run; (left) evaluation lap with the basic example reward function, (right) faster lap with modified reward function

In Figure 17-19 (left), it seems like the car has a small zigzag pattern at times, so one improvement here could be that we penalize the car when it turns too much. In the code example that follows, we multiply the reward by a factor of 0.8, if the car is steering beyond a threshold. We also incentivize the car to go faster by giving 20% more reward if the car goes at 2 m/s or more. When trained with this new reward function, we can see that the car does go faster than the previous reward function. Figure 17-19 (right) shows more stability; the car follows the centerline almost to perfection and finishes the lap roughly two seconds faster than our first attempt. This is just a glimpse in to improving the model. We can continue to iterate on our models and clock better lap times using these tools. Al

of the suggestions are incorporated in to the reward function example shown here:

```
def reward_function(params):
    '''
    Example of penalize steering, which helps mitigate zigzag behaviors and
    speed incentive
    '''

    # Read input parameters
    distance_from_center = params['distance_from_center']
    track_width = params['track_width']
    steering = abs(params['steering_angle']) # Only need absolute steering angle
    speed = params['speed'] # in meter/sec

    # Calculate 3 markers that are at varying distances away from the center line
    marker_1 = 0.1 * track_width
    marker_2 = 0.25 * track_width
    marker_3 = 0.5 * track_width

    # Give higher reward if the agent is closer to the center line and vice versa
    if distance_from_center <= marker_1:
        reward = 1
    elif distance_from_center <= marker_2:
        reward = 0.5
    elif distance_from_center <= marker_3:
        reward = 0.1
    else:
        reward = 1e-3 # likely crashed/ close to off track

    # Steering penalty threshold, change the number based on your action space
      setting
    ABS_STEERING_THRESHOLD = 15

    # Penalize reward if the agent is steering too much
    if steering > ABS_STEERING_THRESHOLD:
        reward *= 0.8

    # Incentivize going faster
    if speed >= 2:
        reward *= 1.2
```

of the suggestions are incorporated in to the reward function example shown here:

```
def reward_function(params):
    '''
    Example of penalize steering, which helps mitigate zigzag behaviors and
    speed incentive
    '''

    # Read input parameters
    distance_from_center = params['distance_from_center']
    track_width = params['track_width']
    steering = abs(params['steering_angle']) # Only need absolute steering angle
    speed = params['speed'] # in meter/sec

    # Calculate 3 markers that are at varying distances away from the center line
    marker_1 = 0.1 * track_width
    marker_2 = 0.25 * track_width
    marker_3 = 0.5 * track_width

    # Give higher reward if the agent is closer to the center line and vice versa
    if distance_from_center <= marker_1:
        reward = 1
    elif distance_from_center <= marker_2:
        reward = 0.5
    elif distance_from_center <= marker_3:
        reward = 0.1
    else:
        reward = 1e-3 # likely crashed/ close to off track

    # Steering penalty threshold, change the number based on your action space
      setting
    ABS_STEERING_THRESHOLD = 15

    # Penalize reward if the agent is steering too much
    if steering > ABS_STEERING_THRESHOLD:
        reward *= 0.8

    # Incentivize going faster
    if speed >= 2:
        reward *= 1.2
```

Racing the AWS DeepRacer Car

It’s time to bring what we’ve learned so far from the virtual to the physical worl and race a real autonomous car. Toy sized, of course!

If you own an AWS DeepRacer car, follow the instructions provided to test your model on a physical car. For those interested in buying the car, AWS DeepRacer is available for purchase on Amazon.

Racing the AWS DeepRacer Car

It’s time to bring what we’ve learned so far from the virtual to the physical world and race a real autonomous car. Toy sized, of course!

If you own an AWS DeepRacer car, follow the instructions provided to test your model on a physical car. For those interested in buying the car, AWS DeepRacer is available for purchase on Amazon.

Building the Track

Now that we have a trained model, we can evaluate this model on a real track and with a physical AWS DeepRacer car. First, let's build a makeshift track at home to race our model. For simplicity, we'll build only part of a racetrack, but instructions on how to build an entire track are provided here.

To build a track, you need the following materials:

For track borders:

> We can create a track with tape that is about two-inches wide and white or off-white color against the dark-colored track surface. In the virtual environment, the thickness of the track markers is set to be two inches. For a dark surface, use a white or off-white tape. For example, 1.88-inch width, pearl white duct tape or 1.88-inch (less sticky) masking tape.

For track surface:

> We can create a track on a dark-colored hard floor such as hardwood, carpet, concrete, or asphalt felt. The latter mimics the real-world road surface with minimal reflection.

Building the Track

Now that we have a trained model, we can evaluate this model on a real track and with a physical AWS DeepRacer car. First, let's build a makeshift track at home to race our model. For simplicity, we'll build only part of a racetrack, but instructions on how to build an entire track are provided here.

To build a track, you need the following materials:

For track borders:

> We can create a track with tape that is about two-inches wide and white or off-white color against the dark-colored track surface. In the virtual environment, the thickness of the track markers is set to be two inches. For a dark surface, use a white or off-white tape. For example, 1.88-inch width, pearl white duct tape or 1.88-inch (less sticky) masking tape.

For track surface:

> We can create a track on a dark-colored hard floor such as hardwood, carpet, concrete, or asphalt felt. The latter mimics the real-world road surface with minimal reflection.

AWS DeepRacer Single-Turn Track Template

This basic track template consists of two straight track segments connected by a curved track segment, as illustrated in Figure 17-20. Models trained with this track should make our AWS DeepRacer vehicle drive in a straight line or make turns in one direction. *The angular dimensions for the turns specified are suggestive; we can use approximate measurements when laying down the track.*

Figure 17-20. The test track layout

AWS DeepRacer Single-Turn Track Template

This basic track template consists of two straight track segments connected by a curved track segment, as illustrated in Figure 17-20. Models trained with this track should make our AWS DeepRacer vehicle drive in a straight line or make turns in one direction. *The angular dimensions for the turns specified are suggestive; we can use approximate measurements when laying down the track.*

Figure 17-20. The test track layout

Running the Model on AWS DeepRacer

To start the AWS DeepRacer vehicle on autonomous driving, we must upload at least one AWS DeepRacer model to our AWS DeepRacer vehicle.

To upload a model, pick our trained model from the AWS DeepRacer console and then download the model artifacts from its Amazon S3 storage to a (local or network) drive that can be accessed from the computer. There's an easy download model button provided on the model page.

To upload a trained model to the vehicle, do the following:

1. From the device console's main navigation pane, choose Models, as shown in Figure 17-21.

Figure 17-21. The Model upload menu on the AWS DeepRacer car web console

2. On the Models page, above the Models list, choose Upload.
3. From the file picker, navigate to the drive or share where you downloaded the model artifacts and choose the model for upload.
4. When the model is uploaded successfully, it will be added to the Models lis and can be loaded into the vehicle's inference engine.

Running the Model on AWS DeepRacer

To start the AWS DeepRacer vehicle on autonomous driving, we must upload at least one AWS DeepRacer model to our AWS DeepRacer vehicle.

To upload a model, pick our trained model from the AWS DeepRacer console and then download the model artifacts from its Amazon S3 storage to a (local or network) drive that can be accessed from the computer. There's an easy download model button provided on the model page.

To upload a trained model to the vehicle, do the following:

1. From the device console's main navigation pane, choose Models, as shown in Figure 17-21.

Figure 17-21. The Model upload menu on the AWS DeepRacer car web console

2. On the Models page, above the Models list, choose Upload.
3. From the file picker, navigate to the drive or share where you downloaded the model artifacts and choose the model for upload.
4. When the model is uploaded successfully, it will be added to the Models list and can be loaded into the vehicle's inference engine.

Driving the AWS DeepRacer Vehicle Autonomously

To start autonomous driving, place the vehicle on a physical track and do the following:

1. Follow the instructions to sign in to the vehicle's device console, and then do the following for autonomous driving:

    a. On the "Control vehicle" page, in the Controls section, choose "Autonomous driving," as shown in Figure 17-22.

Figure 17-22. Driving mode selection menu on the AWS DeepRacer car web console

2. On the "Select a model" drop-down list (Figure 17-23), choose an uploaded model, and then choose "Load model." This will start loading the model into the inference engine. The process takes about 10 seconds to complete.

3. Adjust the "Maximum speed" setting of the vehicle to be a percentage of the maximum speed used in training the model. (Certain factors such as surface friction of the real track can reduce the maximum speed of the vehicle from the maximum speed used in the training. You'll need to experiment to find the optimal setting.)

Figure 17-23. Model selection menu on AWS DeepRacer car web console

4. Choose “Start vehicle” to set the vehicle to drive autonomously.
5. Watch the vehicle drive on the physical track or the streaming video player on the device console.
6. To stop the vehicle, choose “Stop vehicle.”

Sim2Real transfer

Simulators can be more convenient than the real world to train on. Certain scenarios can be easily created in a simulator; for example, a car colliding with a person or another car—we wouldn’t want to do this in the real world :). However, the simulator in most cases will not have the visual fidelity we do in the real world. Also, it might not be able to capture the physics of the real world. These factors can affect modeling the environment in the simulator, and as a result, even with great success in simulation, when the agent runs in the real world, we can experience failures. Here are some of the common approaches to handle the limitations of the simulator:

System identification

Build a mathematical model of the real environment and calibrate the physical system to be as realistic as possible.

Domain adaptation

Map the simulation domain to the real environment, or vice versa, using techniques such as regularization, GANs, or transfer learning.

Domain randomization

Create a variety of simulation environments with randomized properties and train a model on data from all these environments.

In the context of DeepRacer, the simulation fidelity is an approximate representation of the real world, and the physics engine could use some improvements to adequately model all the possible physics of the car. But the beauty of deep reinforcement learning is that we don't need everything to be perfect. To mitigate large perceptual changes affecting the car, we do two major things: a) instead of using an RGB image, we grayscale the image to make the perceptual differences between the simulator and the real world narrower, and b) we intentionally use a shallow feature embedder; for instance, we use only a few CNN layers; this helps the network not learn the simulation environment entirely. Instead it forces the network to learn only the important features. For example, on the race track the car learns to focus on navigating using the white track extremity markers. Take a look at Figure 17-24, which uses a technique called GradCAM to generate a heatmap of the most influential parts of the image to understand where the car is looking for navigation.

Figure 17-24. GradCAM heatmaps for AWS DeepRacer navigation

Domain randomization

> Create a variety of simulation environments with randomized properties and train a model on data from all these environments.

In the context of DeepRacer, the simulation fidelity is an approximate representation of the real world, and the physics engine could use some improvements to adequately model all the possible physics of the car. But the beauty of deep reinforcement learning is that we don't need everything to be perfect. To mitigate large perceptual changes affecting the car, we do two major things: a) instead of using an RGB image, we grayscale the image to make the perceptual differences between the simulator and the real world narrower, and b) we intentionally use a shallow feature embedder; for instance, we use only a few CNN layers; this helps the network not learn the simulation environment entirely. Instead it forces the network to learn only the important features. For example, on the race track the car learns to focus on navigating using the white track extremity markers. Take a look at Figure 17-24, which uses a technique called GradCAM to generate a heatmap of the most influential parts of the image to understand where the car is looking for navigation.

Figure 17-24. GradCAM heatmaps for AWS DeepRacer navigation

Further Exploration

To continue with the adventure, you can become involved in various virtual and physical racing leagues. Following are a few options to explore.

Further Exploration

To continue with the adventure, you can become involved in various virtual and physical racing leagues. Following are a few options to explore.

DeepRacer League

AWS DeepRacer has a physical league at AWS summits and monthly virtual leagues. To race in the current virtual track and win prizes, visit the league page: *https://console.aws.amazon.com/deepracer/home?region=us-east-1#leaderboards*.

DeepRacer League

AWS DeepRacer has a physical league at AWS summits and monthly virtual leagues. To race in the current virtual track and win prizes, visit the league page: *https://console.aws.amazon.com/deepracer/home?region=us-east-1#leaderboards*.

Advanced AWS DeepRacer

We understand that some advanced developers might want more control over the simulation environment and also have the ability to define different neural network architectures. To enable this experience, we provide a Jupyter Notebook–based setup with which you can provision the components needed to train your custom AWS DeepRacer models.

Advanced AWS DeepRacer

We understand that some advanced developers might want more control over the simulation environment and also have the ability to define different neural network architectures. To enable this experience, we provide a Jupyter Notebook–based setup with which you can provision the components needed to train your custom AWS DeepRacer models.

AI Driving Olympics

At NeurIPS 2018, the AI Driving Olympics (AI-DO) was launched with a focus on AI for self-driving cars. During the first edition, this global competition featured lane-following and fleet-management challenges on the Duckietown platform (Figure 17-25). This platform has two components, Duckiebots (miniature autonomous taxis), and Duckietowns (miniature city environments featuring roads, signs). Duckiebots have one job—to transport the citizens of Duckietown (the duckies). Onboard with a tiny camera and running the computations on Raspberry Pi, Duckiebots come with easy-to-use software, helping everyone from high schoolers to university researchers to get their code running relatively quickly. Since the first edition of the AI-DO, this competition has expanded to other top AI academic conferences and now includes challenges featuring both Duckietown and DeepRacer platforms.

Figure 17-25. Duckietown at the AI Driving Olympics

5

DIY Robocars

The DIY Robocars Meetup, initially started in Oakland (California), now has expanded to more than 50 meetup groups around the world. These are fun and engaging communities to try and work with other enthusiasts in the self-driving and autonomous robocar space. Many of them run monthly races and are great venues to physically race your robocar.

DIY Robocars

The DIY Robocars Meetup, initially started in Oakland (California), now has expanded to more than 50 meetup groups around the world. These are fun and engaging communities to try and work with other enthusiasts in the self-driving and autonomous robocar space. Many of them run monthly races and are great venues to physically race your robocar.

Roborace

It’s time to channel our inner Michael Schumacher. Motorsports is typically considered a competition of horsepower, and now Roborace is turning it into a competition of intelligence. Roborace organizes competitions between all-electric, self-driving race cars, like the sleek-looking Robocar designed by Daniel Simon (known for futuristic designs like the Tron Legacy Light Cycle), shown in Figure 17-26. We are not talking about miniature-scaled cars anymore. These are full-sized, 1,350 kg, 4.8 meters, capable of reaching 200 mph (320 kph). The best part? We don’t need intricate hardware knowledge to race them.

The real stars here are AI developers. Each team gets an identical car, so that the winning edge is focused on the autonomous AI software written by the competitors. Output of onboard sensors such as cameras, radar, lidar, sonar, and ultrasonic sensors is available. For high-throughput computations, the cars are also loaded with the powerful NVIDIA DRIVE platform capable of processing several teraflops per second. All we need to build is the algorithm to let the car stay on track, avoid getting into an accident, and of course, get ahead as fast as possible.

To get started, Roborace provides a race simulation environment where precise virtual models of real cars are available. Along comes virtual qualification rounds, where the winners get to participate in real races including at Formula E circuits. By 2019, the top racing teams have already closed the gap to within 5–10% of the best human performances in races. Soon developers will be standing on top of the podium, after beating professional drivers. Ultimately, competitions like these lead to innovations and hopefully the learnings can be translated back to the autonomous car industry, making them safer, more reliable, and performant at the same time.

FROM THE CREATOR’S DESK

By Anima Anandkumar, director of research, NVIDIA, Bren Professor, California Institute of Technology

Future and Evolution of Reinforcement Learning

We would not see reinforcement learning in the usual sense where learning is assumed to be from scratch with no prior knowledge or structure. Instead, I like to use the term “interactive learning,” which is broader. This includes active learning where data collection is done actively based on limitations of the current model that is being trained. I envision the use of physics-infused learning in rob6tics in which we incorporate physics and classical control but add learning to improve efficiency while maintaining stability and safety.

Figure 17-26. Robocar from Roborace designed by Daniel Simon (image courtesy of Roborace)

S

We would not see reinforcement learning in the usual sense where learning is assumed to be from scratch with no prior knowledge or structure. Instead, I like to use the term “interactive learning,” which is broader. This includes active learning where data collection is done actively based on limitations of the current model that is being trained. I envision the use of physics-infused learning in rob6tics in which we incorporate physics and classical control but add learning to improve efficiency while maintaining stability and safety.

Figure 17-26. Robocar from Roborace designed by Daniel Simon (image courtesy of Roborace)

Summary

In the previous chapter, we looked at how to train a model for an autonomous vehicle by manually driving inside the simulator. In this chapter, we explored concepts related to reinforcement learning and learned how to formulate the ultimate problem on everyone's mind: how to get an autonomous car to learn how to drive. We used the magic of reinforcement learning to remove the human from the loop, teaching the car to drive in a simulator on its own. But why limit it to the virtual world? We brought the learnings into the physical world and raced a real car. And all it took was an hour!

Deep reinforcement learning is a relatively new field but an exciting one to explore further. Some of the recent extensions to reinforcement learning are opening up new problem domains where its application can bring about large-scale automation. For example, hierarchical reinforcement learning allows us to model fine-grained decision making. Meta-RL allows us to model generalized decision making across environments. These frameworks are getting us closer to mimicking human-like behavior. It's no surprise that a lot of machine learning researchers believe reinforcement learning has the potential to get us closer to *artificial general intelligence* and open up avenues that were previously considered science fiction.

Summary

In the previous chapter, we looked at how to train a model for an autonomous vehicle by manually driving inside the simulator. In this chapter, we explored concepts related to reinforcement learning and learned how to formulate the ultimate problem on everyone's mind: how to get an autonomous car to learn how to drive. We used the magic of reinforcement learning to remove the human from the loop, teaching the car to drive in a simulator on its own. But why limit it to the virtual world? We brought the learnings into the physical world and raced a real car. And all it took was an hour!

Deep reinforcement learning is a relatively new field but an exciting one to explore further. Some of the recent extensions to reinforcement learning are opening up new problem domains where its application can bring about large-scale automation. For example, hierarchical reinforcement learning allows us to model fine-grained decision making. Meta-RL allows us to model generalized decision making across environments. These frameworks are getting us closer to mimicking human-like behavior. It's no surprise that a lot of machine learning researchers believe reinforcement learning has the potential to get us closer to *artificial general intelligence* and open up avenues that were previously considered science fiction.

Appendix A. A Crash Course in Convolutional Neural Networks

In keeping with the word “practical” in the book’s title, we’ve focused heavily on the real-world aspects of deep learning. The goal of this appendix is meant to serve as reference material, rather than a full-fledged exploration into the theoretical aspects of deep learning. To develop a deeper understanding of some of these topics, we recommend perusing the “Further Exploration” for references to other source material.

Appendix A. A Crash Course in Convolutional Neural Networks

In keeping with the word “practical” in the book’s title, we’ve focused heavily on the real-world aspects of deep learning. The goal of this appendix is meant to serve as reference material, rather than a full-fledged exploration into the theoretical aspects of deep learning. To develop a deeper understanding of some of these topics, we recommend perusing the “Further Exploration” for references to other source material.

Machine Learning

- Machine learning helps learn patterns from data to make predictions on unseen data.
- There are three kinds of machine learning: supervised learning (learning from labeled data), unsupervised learning (learning from unlabeled data), and reinforcement learning (learning by action and feedback from an environment).
- Supervised learning tasks include classification (output is one of many categories/classes) and regression (output is a numeric value).
- There are various supervised machine learning techniques including naive Bayes, SVM, decision trees, k-nearest neighbors, neural networks, and others.

Machine Learning

- Machine learning helps learn patterns from data to make predictions on unseen data.
- There are three kinds of machine learning: supervised learning (learning from labeled data), unsupervised learning (learning from unlabeled data), and reinforcement learning (learning by action and feedback from an environment).
- Supervised learning tasks include classification (output is one of many categories/classes) and regression (output is a numeric value).
- There are various supervised machine learning techniques including naive Bayes, SVM, decision trees, k-nearest neighbors, neural networks, and others.

Perceptron

- A perceptron, as shown in Figure A-1, is the simplest form of a neural network, a single-layered neural network with one neuron.
- A perceptron calculates a weighted sum of its inputs; that is, it accepts inpu values, multiplies each with a corresponding weight, adds a bias term, and generates a numeric output.

Figure A-1. An example of a perceptron

- Because a perceptron is governed by a linear equation, it only has the capacity to model linear or close-to-linear tasks well. For a regression task, the prediction can be represented as a straight line. For a classification task, the prediction can be represented as a straight line separating a plane into two parts.
- Most practical tasks are nonlinear in nature, and hence a perceptron would fail to model the underlying data well, leading to poor prediction performance.

t

Activation Functions

- Activation functions convert linear input into nonlinear output. A sigmoid (Figure A-2) is a type of activation function. The hyperbolic tangent function and ReLU, shown in Figure A-3 are other common activation functions.

Figure A-2. A plot of the sigmoid function

Figure A-3. A plot of the ReLU function

- For classification, the probability of predicting a category is usually desired This can be accomplished by a sigmoid function (for binary classification tasks) at the end of the perceptron, which converts values in the range of 0 to 1, and hence is better suited to represent probabilities.
- Activation functions help replicate the biological concept of the firing of neurons, which, in the simplest case, can be on or off; that is, the inputs activate the neurons. A step function is one form of an activation function where the value is one when the neuron is activated and zero when not. One downside of using a step function is that it loses the magnitude. ReLU, on the other hand, outputs the magnitude of the activation if it activates, and zero otherwise. Due to its simplicity and low computational requirements, i is preferred for nonprobability-based outputs.

Figure A-3. A plot of the ReLU function

- For classification, the probability of predicting a category is usually desired. This can be accomplished by a sigmoid function (for binary classification tasks) at the end of the perceptron, which converts values in the range of 0 to 1, and hence is better suited to represent probabilities.
- Activation functions help replicate the biological concept of the firing of neurons, which, in the simplest case, can be on or off; that is, the inputs activate the neurons. A step function is one form of an activation function where the value is one when the neuron is activated and zero when not. One downside of using a step function is that it loses the magnitude. ReLU, on the other hand, outputs the magnitude of the activation if it activates, and zero otherwise. Due to its simplicity and low computational requirements, it is preferred for nonprobability-based outputs.

Neural Networks

- Combining a perceptron with a nonlinear activation function improves its predictive ability. Combining several perceptions with nonlinear activation functions improves it even further.
- A neural network, as seen in Figure A-4, consists of multiple layers, each containing multiple perceptrons. Layers are connected in a sequence, passing the output of the previous layer as the input to the subsequent layer. This data transfer is represented as connections, with each neuron connected to every neuron in the previous and subsequent layers (hence the layers are aptly named fully connected layers). Each of these connections has a multiplying factor associated with it, also known as a weight.

Figure A-4. An example of a multilayer neural network

- Each neuron stores the associated weights and bias, and has an optional nonlinear activation function at the end of it.
- Since information flows in one direction, without any cycles, this network structure is also called a feedforward neural network or multilayer feedforward network.
- The layers between the input(s) and output(s), hidden from the end-user, are appropriately known as hidden layers.
- Neural networks have a powerful ability to model patterns in data (i.e., represent the underlying relationship between data and associated labels).

The *Universal Approximation Theorem* states that a neural network with a single hidden layer can represent any continuous function within a specific range, acting as a universal approximator. So technically, a single-layer network could model any problem in existence. The major caveat is that it might be infeasible in practice due to the sheer number of neurons required to model many problems. The alternative approach is to have multiple layers with a fewer number of neurons per layer, which works well in practice.

- The output of a single hidden layer is the result of a nonlinear function applied over a linear combination of its inputs. The output of a second hidden layer is the result of a nonlinear function applied over a linear combination of inputs, which itself are nonlinear functions applied over a linear combination of their inputs. Each layer further builds upon the predictive power of the previous layer. Thus, the higher the number of layers, the better the representative ability of the neural network to model real-world nonlinear problems. This makes the network deep, and hence such networks with many hidden layers are also called *deep neural networks*.

The *Universal Approximation Theorem* states that a neural network with a single hidden layer can represent any continuous function within a specific range, acting as a universal approximator. So technically, a single-layer network could model any problem in existence. The major caveat is that it might be infeasible in practice due to the sheer number of neurons required to model many problems. The alternative approach is to have multiple layers with a fewer number of neurons per layer, which works well in practice.

- The output of a single hidden layer is the result of a nonlinear function applied over a linear combination of its inputs. The output of a second hidden layer is the result of a nonlinear function applied over a linear combination of inputs, which itself are nonlinear functions applied over a linear combination of their inputs. Each layer further builds upon the predictive power of the previous layer. Thus, the higher the number of layers, the better the representative ability of the neural network to model real-world nonlinear problems. This makes the network deep, and hence such networks with many hidden layers are also called *deep neural networks*.

Backpropagation

- Training a network involves modifying the weights and biases of the network iteratively until the network starts to make predictions with minimal error.
- Backpropagation helps train neural networks. It involves making a prediction, measuring how far-off the prediction is from the ground truth, and propagating the error back into the network so that the weights can be updated in order to reduce the magnitude of the error in subsequent iterations. This process is repeated until the error of the network no longer reduces.
- Neural networks are usually initialized with random weights.
- The performance of a model is measured with loss functions. For regression tasks (with numeric predictions), mean squared error (MSE) is a commonly used loss function. Due to the squaring of the magnitude of the error, using MSE helps favor more small errors than a few large errors. For classification tasks (with category predictions), the cross-entropy loss is commonly used. While accuracy is used more as a human-understandable metric of performance, cross-entropy loss is used for training a network, since it’s more nuanced.

Backpropagation

- Training a network involves modifying the weights and biases of the network iteratively until the network starts to make predictions with minimal error.
- Backpropagation helps train neural networks. It involves making a prediction, measuring how far-off the prediction is from the ground truth, and propagating the error back into the network so that the weights can be updated in order to reduce the magnitude of the error in subsequent iterations. This process is repeated until the error of the network no longer reduces.
- Neural networks are usually initialized with random weights.
- The performance of a model is measured with loss functions. For regression tasks (with numeric predictions), mean squared error (MSE) is a commonly used loss function. Due to the squaring of the magnitude of the error, using MSE helps favor more small errors than a few large errors. For classification tasks (with category predictions), the cross-entropy loss is commonly used. While accuracy is used more as a human-understandable metric of performance, cross-entropy loss is used for training a network, since it’s more nuanced.

Shortcoming of Neural Networks

- A layer of a neural network with `n` neurons and `i` inputs requires `n x i` weights and computations since each neuron is connected to its inputs. An image usually has a lot of pixels as inputs. For an image with width `w`, height `h`, and `c` channels, `w x h x c x n` computations would be required. For a standard smartphone camera's image (usually 4032 x 3024 pixels in 3 channels of red, green, and blue), if the first hidden layer has 128 neurons, over 4 billion weights and computations would result, making neural networks impractical to use for most computer-vision tasks. We want to ideally convert an image into a much smaller workable representation that can then be classified by a neural network.
- Pixels in an image usually are related to other pixels in their vicinity. Capturing this relationship can give a better understanding of the image's content. But while feeding a neural network, the relationship between these pixels is lost because the image is flattened out into a 1D array.

Shortcoming of Neural Networks

- A layer of a neural network with `n` neurons and `i` inputs requires `n x i` weights and computations since each neuron is connected to its inputs. An image usually has a lot of pixels as inputs. For an image with width `w`, height `h`, and `c` channels, `w x h x c x n` computations would be required. For a standard smartphone camera's image (usually 4032 x 3024 pixels in 3 channels of red, green, and blue), if the first hidden layer has 128 neurons, over 4 billion weights and computations would result, making neural networks impractical to use for most computer-vision tasks. We want to ideally convert an image into a much smaller workable representation that can then be classified by a neural network.

- Pixels in an image usually are related to other pixels in their vicinity. Capturing this relationship can give a better understanding of the image's content. But while feeding a neural network, the relationship between these pixels is lost because the image is flattened out into a 1D array.

Desired Properties of an Image Classifier

- It is computationally efficient.
- It captures relationships between different parts of an image.
- It is resilient to differences in position, size, angle of an object present in an image, and other factors such as noise. That is, an image classifier should be invariant to geometric transformations (often referred to using terms such as translation invariance, rotational invariance, positional invariance, shift-invariance, scale invariance, etc.).
- A CNN satisfies many of these desired properties in practice (some by its design, others with data augmentation).

Desired Properties of an Image Classifier

- It is computationally efficient.
- It captures relationships between different parts of an image.
- It is resilient to differences in position, size, angle of an object present in an image, and other factors such as noise. That is, an image classifier should be invariant to geometric transformations (often referred to using terms such as translation invariance, rotational invariance, positional invariance, shift-invariance, scale invariance, etc.).
- A CNN satisfies many of these desired properties in practice (some by its design, others with data augmentation).

Convolution

- A convolutional filter is a matrix that slides across an image (row by row, left to right), and multiplies with that section of adjoining pixels to produce an output. Such filters are famously used in Photoshop to blur, sharpen, brighten, darken an image, etc.
- Convolutional filters can, among other things, be used to detect vertical, horizontal, and diagonal edges. That is, when edges are present, they generate higher numbers as output, and lower output values otherwise. Alternatively stated, they “filter” out the input unless it contains an edge.
- Historically, convolutional filters were hand-engineered to perform a task. In the context of neural networks, these convolutional filters can be used as building blocks whose parameters (matrix weights) can be deduced automatically via a training pipeline.
- A convolutional filter activates every time it spots the input pattern it has been trained for. Since it traverses over an entire image, the filter is capable of locating the pattern at any position in the image. As a result, the convolutional filters contribute to making the classifier position invariant.

Convolution

- A convolutional filter is a matrix that slides across an image (row by row, left to right), and multiplies with that section of adjoining pixels to produce an output. Such filters are famously used in Photoshop to blur, sharpen, brighten, darken an image, etc.
- Convolutional filters can, among other things, be used to detect vertical, horizontal, and diagonal edges. That is, when edges are present, they generate higher numbers as output, and lower output values otherwise. Alternatively stated, they “filter” out the input unless it contains an edge.
- Historically, convolutional filters were hand-engineered to perform a task. In the context of neural networks, these convolutional filters can be used as building blocks whose parameters (matrix weights) can be deduced automatically via a training pipeline.
- A convolutional filter activates every time it spots the input pattern it has been trained for. Since it traverses over an entire image, the filter is capable of locating the pattern at any position in the image. As a result, the convolutional filters contribute to making the classifier position invariant.

Pooling

- A pooling operation reduces the spatial size of the input. It looks at smaller portions of an input and performs an aggregation operation reducing the output to a single number. Typically aggregation operations include finding the maximum or average of all values, also commonly known as *max pooling* and *average pooling*, respectively. For example, a `2 x 2` max pooling operation will replace each `2 x 2` submatrix of the input (without overlap) with a single value in the output. Similar to convolution, pooling traverses the entire input matrix, aggregating multiple blocks of values into single values. If the input matrix were of size `2N x 2N`, the resulting output would be of size `N x N`.
- In the aforementioned example, the output would be the same regardless of where in the `2 x 2` matrix the maximum occurred.
- Max pooling is more commonly used than average pooling because it acts as a noise suppressor. Because it only takes the maximum value in each operation, it ignores the lower values, which are typically characteristics of noise.

Pooling

- A pooling operation reduces the spatial size of the input. It looks at smaller portions of an input and performs an aggregation operation reducing the output to a single number. Typically aggregation operations include finding the maximum or average of all values, also commonly known as *max pooling* and *average pooling*, respectively. For example, a `2 x 2` max pooling operation will replace each `2 x 2` submatrix of the input (without overlap) with a single value in the output. Similar to convolution, pooling traverses the entire input matrix, aggregating multiple blocks of values into single values. If the input matrix were of size `2N x 2N`, the resulting output would be of size `N x N`.
- In the aforementioned example, the output would be the same regardless of where in the `2 x 2` matrix the maximum occurred.
- Max pooling is more commonly used than average pooling because it acts as a noise suppressor. Because it only takes the maximum value in each operation, it ignores the lower values, which are typically characteristics of noise.

Structure of a CNN

- At a high level, a CNN (shown in Figure A-5) consists of two parts—a featurizer followed by a classifier. The featurizer reduces the input image to a workable size of features (i.e., a smaller representation of the image) upon which the classifier then acts.
- A CNN's featurizer is a repeating structure of convolutional, activation, and pooling layers. The input passes through this repeating structure in a sequence one layer at a time.

Figure A-5. A high-level overview of a convolutional network

- The transformed input is finally passed into classification layers, which are really just neural networks. In this context, the classification neural network is typically known as a fully connected layer.
- Much like in neural networks, the nonlinearity provided by activation functions help the CNN learn complex features. ReLU is the most commonly used activation function.

- Empirically, earlier layers (i.e., those closer to the input layers) detect simple features such as curves and edges. If a matching feature is detected, the layer gets activated and passes on the signal to subsequent layers. Eventual layers combine these signals to detect more complex features such as a human eye, human nose, or human ears. Ultimately, these signals might add up to detect the entire object such as a human face.
- CNNs are trained using backpropagation just like multilayer perceptron networks.
- Unlike traditional machine learning, this makes feature selection automated. Additionally, the reduction of the input into smaller features makes the process more computationally efficient. Both of these are big selling points of deep learning.

- Empirically, earlier layers (i.e., those closer to the input layers) detect simple features such as curves and edges. If a matching feature is detected, the layer gets activated and passes on the signal to subsequent layers. Eventual layers combine these signals to detect more complex features such as a human eye, human nose, or human ears. Ultimately, these signals might add up to detect the entire object such as a human face.
- CNNs are trained using backpropagation just like multilayer perceptron networks.
- Unlike traditional machine learning, this makes feature selection automated. Additionally, the reduction of the input into smaller features makes the process more computationally efficient. Both of these are big selling points of deep learning.

Further Exploration

The topics we cover in this book should be self-sufficient, but if you do choose to develop a deeper appreciation for the underlying fundamentals, we highly recommend checking out some of the following material:

https://www.deeplearning.ai

This set of deep learning courses from Andrew Ng covers the foundations fo a wide range of topics, encompassing theoretical knowledge and deep insights behind each concept.

https://course.fast.ai

This course by Jeremy Howard and Rachel Thomas has a more hands-on approach to deep learning using PyTorch as the primary deep learning framework. The fast.ai library used in this course has helped many students run models with state-of-the-art performance using just a few lines of code.

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition

This book by Aurélien Géron covers machine learning and delves into topics such as support vector machines, decision trees, random forests, etc., then bridges into deep learning topics such as convolutional and recurrent neural networks. It explains both theory as well as some hands-on examples of these techniques.

Further Exploration

The topics we cover in this book should be self-sufficient, but if you do choose to develop a deeper appreciation for the underlying fundamentals, we highly recommend checking out some of the following material:

https://www.deeplearning.ai

This set of deep learning courses from Andrew Ng covers the foundations for a wide range of topics, encompassing theoretical knowledge and deep insights behind each concept.

https://course.fast.ai

This course by Jeremy Howard and Rachel Thomas has a more hands-on approach to deep learning using PyTorch as the primary deep learning framework. The fast.ai library used in this course has helped many students run models with state-of-the-art performance using just a few lines of code.

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition

This book by Aurélien Géron covers machine learning and delves into topics such as support vector machines, decision trees, random forests, etc., then bridges into deep learning topics such as convolutional and recurrent neural networks. It explains both theory as well as some hands-on examples of these techniques.

Index

Symbols

24/7 availability, Real-Time Object Classification on iOS with Core ML

80legs.com, Data

A

A/B testing, A/B Testing, A/B Testing Hosted Models-Using the Experiment in Code

accountability, Accountability and Explainability

action space, Configure the action space-Configure the action space

activation functions, Activation Functions

actor-critic approach, Policy based or value based—why not both?

Adam optimizer, Set Training Parameters

Advanced Driver Assistance Systems (ADAS), Autonomous Cars

Advanced Virtual RISC (AVR), Arduino

Advantage function, Policy based or value based—why not both?

AI Driving Olympics (AI-DO), AI Driving Olympics

Airbnb, Airbnb's Photo Classification

AirSim, Deep Learning, Autonomous Driving, and the Data Problem, Setup and Requirements

AlchemyVision, IBM Watson Visual Recognition

Algorithmia, Algorithmia

Alibaba, Speaker Recognition by Alibaba

alpha users, The Life Cycle of a Food Classifier App

Amazon Rekognition, Amazon Rekognition

Amazon SageMaker, A Brief Introduction to Reinforcement Learning

AMD GPUs, Installation

Anaconda Python, Installation

Android Studio, Building a Real-Time Object Recognition App

annotations, Data

Annoy (Approximate Nearest Neighbors Oh Yeah), Annoy

Apollo simulation, Deep Learning, Autonomous Driving, and the Data Problem

app size, reducing, Reducing App Size-Use Create ML

Apple's machine learning architecture, Apple's Machine Learning Architecture-ML Performance Primitives, Approach 2: Use Create ML

approximate nearest neighbor (ANN), Approximate Nearest-Neighbor Benchmark

architecture, Model Architecture

Arduino, Arduino

artificial general intelligence, Summary

artificial intelligence (AI)

- browser-based, AI in the Browser with TensorFlow.js and ml5.js-Summary
- definition of, What Is AI?
- on embedded devices, Becoming a Maker: Exploring Embedded AI at the Edge-Summary

examples of, An Apology, Motivating Examples-Motivating Examples (see also case studies)

frequently asked questions, Frequently Asked Questions-Frequently Asked Questions

history of, A Brief History of AI-How Deep Learning Became a Thing

versus human accuracy, ImageNet Dataset

introduction to, The Real Introduction-What Is AI?

mobile-based, Real-Time Object Classification on iOS with Core ML-Summary, Shazam for Food: Developing Android Apps with TensorFlow Lite and ML Kit-Summary

perfect deep learning solution, Recipe for the Perfect Deep Learning Solution-Hardware, What's in the Picture: Image Classification with Keras

resources for learning about, Education and Exploration, Further Exploration

responsible AI, Responsible AI-Privacy

serving prediction requests, Landscape of Serving AI Predictions

tools, tips, and tricks, Practical Tools, Tips, and Tricks-One Last Question

aspect ratio, Effect of Change in Aspect Ratio on Transfer Learning

asynchronous communication, Prefetch Data

augmentations, AutoAugment

AutoAugment, Tools of the Trade, AutoAugment

AutoKeras, Tools of the Trade, AutoKeras

Automatic Mixed Precision, Use Automatic Mixed Precision

automatic tuning, AutoAugment

s autonomous driving

data collection necessary for, Deep Learning, Autonomous Driving, and the Data Problem

data exploration and preparation, Data Exploration and Preparation-Dataset Imbalance and Driving Strategies

deep learning applications in, Deep Learning, Autonomous Driving, and the Data Problem

further exploration, Further Exploration-Reinforcement Learning

history of, A Glimmer of Hope, A Brief History of Autonomous Driving

introduction to, Simulating a Self-Driving Car Using End-to-End Deep Learning with Keras

landscape of, Why Learn Reinforcement Learning with an Autonomous Car?

model deployment, Deploying Our Autonomous Driving Model-Deploying Our Autonomous Driving Model

model training, Training Our Autonomous Driving Model-Callbacks

object detection in, Autonomous Cars

real-world applications of, Further Exploration

SAE levels of driving automation, Simulating a Self-Driving Car Using End-to-End Deep Learning with Keras

simulation and, Deep Learning, Autonomous Driving, and the Data Problem, Practical Deep Reinforcement Learning with DeepRacer, Sim2Real transfer

steering through simulated environments, The “Hello, World!” of Autonomous Driving: Steering Through a Simulated Environment-Setup and Requirements

Autonomous Driving Cookbook, Deep Learning, Autonomous Driving, and the Data Problem, Reinforcement Learning

availability, High Availability, Real-Time Object Classification on iOS with Core ML

AWS DeepRacer

for advanced developers, Advanced AWS DeepRacer

AI Driving Olympics, AI Driving Olympics

creating AWS accounts, Building Our First Reinforcement Learning

DeepRacer league, DeepRacer League

DIY Robocars, DIY Robocars

improving learning models, Step 5: Improving Reinforcement Learning Models-Racing the AWS DeepRacer Car

model creation, Step 1: Create Model

model evaluation, Step 4: Evaluating the Performance of the Model

model training, Step 3: Model Training

motivation behind, A Brief Introduction to Reinforcement Learning

racing the car, Racing the AWS DeepRacer Car-Sim2Real transfer

reinforcement learning algorithm, Reinforcement Learning Algorithm in AWS DeepRacer

reinforcement learning in action, How Does a Reinforcement Learning System Learn?

reinforcement learning summary, Deep Reinforcement Learning Summary with DeepRacer as an Example

reinforcement learning terminology, Practical Deep Reinforcement Learning with DeepRacer

reinforcement learning theory, Reinforcement Learning Theory

reward function in, Practical Deep Reinforcement Learning with DeepRacer

Roborace, Roborace

training configuration, Step 2: Configure Training-Configure stop conditions

B

backpropagation, A Glimmer of Hope, Batch Size, Backpropagation

batch size

effect on accuracy, Effect of Batch Size

role of, Batch Size

using larger, Use Larger Batch Size

Bayesian Optimization, Training

BDD100K, Data

benchmarks, Examine Industry Benchmarks

bias, Bias, Class Activation Maps, Image Captioning, Bias-Bias

binary classification, Binary classification

Binder, Installation

Bing Image Search API, Approach 3: Web Scraper Using Bing Image Search API

blind and low-vision community, Face Detection in Seeing AI

bottleneck features, Image Similarity

(see also embeddings)

browser-based AI

benchmarking and practical considerations, Benchmarking and Practical Considerations

case studies, Case Studies-GAN Lab

introduction to, AI in the Browser with TensorFlow.js and ml5.js

JavaScript-based ML libraries, JavaScript-Based Machine Learning Libraries: A Brief History-TensorFlow.js

ml5.js, ml5.js

model conversion, Model Conversion for the Browser

pix2pix, pix2pix-pix2pix

PoseNet, PoseNet

pretrained models using TensorFlow.js, Running Pretrained Models Using TensorFlow.js

TensorFlow.js architecture, TensorFlow.js Architecture

training, Training in the Browser-GPU Utilization

brute-force algorithm, Brute Force

C

caching, Cache Data

callbacks, Callbacks

Carla, Deep Learning, Autonomous Driving, and the Data Problem

case studies

browser-based AI

Airbnb’s photo classification, Airbnb’s Photo Classification

GAN Lab, GAN Lab

Metacar, Metacar

Semi-Conductor, Semi-Conductor

TensorSpace, TensorSpace

computer vision

Giphy, Giphy

InDro Robotics, InDro Robotics

New York Times, The New York Times

OmniEarth, OmniEarth

Photobucket, Photobucket

Staples, Staples

Uber, Uber

embedded AI devices

cucumber sorter, Cucumber Sorter

JetBot, JetBot

squatting for metro tickets, Squatting for Metro Tickets

mobile-based AI

face filter–like functionality, Face Contours in ML Kit

HomeCourt, HomeCourt

InstaSaber, InstaSaber + YoPuppet

Lose It!, Lose It!

Magic Sudoku, Magic Sudoku

portrait mode on Pixel 3 phones, Portrait Mode on Pixel 3 Phones

Seeing AI, Seeing AI

speaker recognition by Alibaba, Speaker Recognition by Alibaba

video segmentation, Real-Time Video Segmentation in YouTube Stories

YoPuppet, InstaSaber + YoPuppet

object detection

autonomous cars, Autonomous Cars

crowd counting, Crowd Counting

face detection in Seeing AI app, Face Detection in Seeing AI

smart refrigerators, Smart Refrigerator

reverse image search

Celebslike.me, Celebrity Doppelgangers

Flickr, Flickr

image captioning, Image Captioning

Pinterest, Pinterest

Spotify, Spotify

categorical_crossentropy loss function, Start Training

category unawareness, ImageNet Dataset

celebrity doppelgangers, Celebrity Doppelgangers

channels, Predicting an Image's Category

chroma keying technique, Real-Time Video Segmentation in YouTube Stories

Clarifai, Clarifai

class activation maps (heat maps), Class Activation Maps-Class Activation Maps, Heatmap visualization

classification problems (see also custom classifiers; image classification)

binary, Binary classification

computer-vision tasks, Classification

multiclass, Multiclass classification

classification with localization, Localization

Cloud AutoML, Building a Custom Detector Without Any Code

cloud computing (see also inference serving)

comparing custom classification APIs, Comparing Custom Classification APIs

example scenario using, Cloud APIs for Computer Vision: Up and Running in 15 Minutes

getting up and running with, Getting Up and Running with Cloud APIs

hosted models, Hosted Models-Using the Experiment in Code

performance tuning for cloud APIs, Performance Tuning for Cloud APIs-Effect of Resizing on OCR APIs

prebuilt object detection APIs, Invoking Prebuilt Cloud-Based Object Detection APIs-Invoking Prebuilt Cloud-Based Object Detection APIs

reduced costs of mobile-based AI, Real-Time Object Classification on iOS with Core ML

training custom classifiers, Training Our Own Custom Classifier-Training Our Own Custom Classifier

troubleshooting classifiers, Top Reasons Why Our Classifier Does Not Work Satisfactorily

visual recognition APIs available, The Landscape of Visual Recognition APIs-What’s unique about this API?

visual recognition APIs comparison, Comparing Visual Recognition APIs-Bias

co-occurrences, Class Activation Maps, Approach 2: Fatkun Chrome Browser Extension

COCO (Microsoft Common Objects in COntext), Datasets, Image Captioning

code examples

- downloading, Using Code Examples
- permission to use, Using Code Examples

code-free training, A continuously evolving landscape, Building a Custom Detector Without Any Code

Cognata, Deep Learning, Autonomous Driving, and the Data Problem

Cognitive Services, Microsoft Cognitive Services, Invoking Prebuilt Cloud-Based Object Detection APIs

CometML, Training

components, Predicting an Image's Category

compression, Performance Tuning for Cloud APIs

computer vision

- case studies, Case Studies-InDro Robotics
- comparing visual recognition APIs, Comparing Visual Recognition APIs-Bias
- custom classifer training, Training Our Own Custom Classifier-Training Our Own Custom Classifier
- custom classification APIs comparison, Comparing Custom Classification APIs-Comparing Custom Classification APIs
- example scenario using, Cloud APIs for Computer Vision: Up and Running in 15 Minutes-Cloud APIs for Computer Vision: Up and Running in 15 Minutes

performance tuning for cloud APIs, Performance Tuning for Cloud APIs-Effect of Resizing on OCR APIs

time estimation for achieving, How Deep Learning Became a Thing

troubleshooting classifiers, Top Reasons Why Our Classifier Does Not Work Satisfactorily

types of computer-vision tasks, Types of Computer-Vision Tasks-Approaches to Object Detection

using cloud APIs, Getting Up and Running with Cloud APIs-Getting Up and Running with Cloud APIs

visual recognition APIs available, The Landscape of Visual Recognition APIs-What's unique about this API?

-

concept drift, Model Versioning

conditional GANs , pix2pix

contact information for this book, How to Contact Us

continuous action space, Configure the action space

contrastive loss function, Siamese Networks for One-Shot Face Verificatior

conventions used in this book, Conventions Used in This Book

ConvNetJS, ConvNetJS

convolution filters, Convolution

Convolutional Neural Networks (CNNs)

accuracy and feature lengths, Length of Feature Vectors

creating diagrams of, Model

end-to-end learning example pipeline, End-to-End Deep Learning Example Pipeline-Basic Custom Network Pipeline

hyperparameter effect on accuracy, How Hyperparameters Affect

Accuracy-Effect of Change in Aspect Ratio on Transfer Learning

image classification using, Predicting an Image's Category, A Shallow Dive into Convolutional Neural Networks

improving model accuracy, From Novice to Master Predictor: Maximizing Convolutional Neural Network Accuracy-From Novice to Master Predictor: Maximizing Convolutional Neural Network Accuracy

progress over past decade, How Deep Learning Became a Thing

resources for learning about, Further Reading-Further Reading

structure of, A Shallow Dive into Convolutional Neural Networks, Structure of a CNN

techniques for ML experimentation, Common Techniques for Machine Learning Experimentation-Reproducible Experiments

tools for automated tuning, Tools to Automate Tuning for Maximum Accuracy-AutoKeras

tools to reduce code and effort, Tools of the Trade-tf-explain

transfer learning using, Transfer Learning

versus transfer learning, Approach 2: Use Create ML

Coral USB accelerator, Google Coral USB Accelerator, Speeding Up with the Google Coral USB Accelerator

Core ML

alternatives to, Alternatives to Core ML

building real-time object recognition apps, Building a Real-Time Object Recognition App-Building a Real-Time Object Recognition App

conversion to, Conversion to Core ML, Model Conversion Using Core ML Tools

dynamic model deployment, Dynamic Model Deployment

ecosystem, Apple’s Machine Learning Architecture-ML Performance Primitives

history of, A Brief History of Core ML

measuring energy consumption, Measuring Energy Impact-Benchmarking Load

Not Hotdog app, Not Hotdog on iOS with Core ML and Create ML-Summary

on-device training, On-Device Training

performance analysis, Performance Analysis-Benchmarking Models on iPhones

reducing app size, Reducing App Size-Use Create ML

counting people, Crowd Counting

Create ML, Use Create ML, Approach 2: Use Create ML-Approach 2: Use Create ML, Building a Custom Detector Without Any Code

crowd counting, Crowd Counting

cucumber sorting device, Cucumber Sorter

curse of dimensionality, Reducing Feature-Length with PCA

custom classifiers

building data pipelines, Build the Data Pipeline

data augmentation, Data Augmentation

hosting and serving (see inference serving)

model analysis, Analyzing the Results-Analyzing the Results

model definition, Model Definition

model testing, Test the Model

model training, Set Training Parameters

organizing data, Organize the Data

overview of, Building a Custom Classifier in Keras with Transfer Learning

using cloud APIs, Training Our Own Custom Classifier-Effect of Resizing on OCR APIs

custom detectors, building without code, Building a Custom Detector Without Any Code-Building a Custom Detector Without Any Code

(see also object detection)

CustomVision.ai, Approach 1: Use Web UI-based Tools-Approach 1: Use Web UI-based Tools, Model Conversion Using Core ML Tools, Building a Custom Detector Without Any Code

D

data augmentation, Data Augmentation-Data Augmentation, Data Augmentation, Data Augmentation

data caching, Cache Data

data generators, Drive Data Generator

data inspection, Data Inspection

data preparation (TensorFlow)

reducing size of input data, Reduce Size of Input Data

storing data as TFRecords, Store as TFRecords

using TensorFlow Datasets, Use TensorFlow Datasets

data reading (TensorFlow)

autotuning parameter values, Autotune Parameter Values

caching data, Cache Data

enabling nondeterministic ordering, Enable Nondeterministic Ordering

parallelizing CPU processing, Parallelize CPU Processing

parallelizing I/O and processing, Parallelize I/O and Processing

prefetching data, Prefetch Data

turning on experimental optimizations, Turn on Experimental Optimizations

using tf.data, Use tf.data

data size

effect on accuracy, Effect of Data Size on Transfer Learning

reducing image size, Performance Tuning for Cloud APIs

reducing in TensorFlow, Reduce Size of Input Data

datamash tool, nvidia-smi

dataset imbalance, Dataset Imbalance and Driving Strategies

Dataset.cache() function, Cache Data

DatasetList.com, Data

datasets

annotating data, Data

approaches to collecting data, Collecting Data-Approach 3: Web Scraper Using Bing Image Search API

collecting hundreds of images, Data, Approach 2: Fatkun Chrome Browser Extension

downloading data in custom ways, Data

finding larger, Data

finding prebuilt, Data

g

finding smaller, Data

for mobile AI app development, How Do I Collect Initial Data?, Data Collection

Google Dataset Search, Data

ImageNet dataset, Predicting an Image's Category, ImageNet Dataset-ImageNet Dataset

largest labeled, Data

MS COCO (Microsoft Common Objects in COntext), Datasets, Image Captioning

need for labeled, Datasets

of negative classes, Data

NIST and MNIST datasets, A Glimmer of Hope

publicly available, Datasets, Approach 1: Find or Collect a Dataset

reading popular, Data

ready-to-use, TensorFlow Datasets

scraping images using Bing Image Search API, Approach 3: Web Scraper Using Bing Image Search API

scraping images using command line, Data

sequential data, Training on Sequential Data

splitting into train, validation, and test, Breaking the Data: Train, Validation, Test

synthetic datasets, Which Library Should I Use?, Data

TensorFlow Datasets, TensorFlow Datasets, Use TensorFlow Datasets, Data

for unique problems, Data

versioning tools for, Data

video datasets, Data

DAVE-2 system, Further Exploration

DAWNBench benchmarks, Examine Industry Benchmarks

debugging, Training

deep learning

applications in autonomous driving, Deep Learning, Autonomous Driving, and the Data Problem

definition of, What Is AI?

highlights of modern era, How Deep Learning Became a Thing

increase in popularity of, How Deep Learning Became a Thing

libraries for, Frameworks

perfect deep learning solution, Recipe for the Perfect Deep Learning Solution-Hardware

Deeplearning.ai, Education and Exploration

detection, Detection (see also facial recognition; object detection)

Digital Data Divide, Data

discount factor (γ), Delayed rewards and discount factor (γ)

discrete action space, Configure the action space

Distributed Deep Reinforcement Learning for Autonomous Driving, Reinforcement Learning

distributed training, Distribute Training

DIY Robocars, DIY Robocars

Docker, Installation

driving automation, Simulating a Self-Driving Car Using End-to-End Deep Learning with Keras

(see also autonomous driving)

E

eager execution, Use tf.function

early stopping, Early Stopping, Callbacks

edge devices (see embedded AI devices)

embedded AI devices

case studies, Case Studies-Cucumber Sorter

comparison of, A Qualitative Comparison of Embedded AI Devices-A Qualitative Comparison of Embedded AI Devices

landscape of, Exploring the Landscape of Embedded AI Devices-Arduino

performance of, Comparing the Performance of Edge Devices

resources for learning about, Further Exploration

embeddings

benefits of, Visualizing Image Clusters with t-SNE

defined, Image Similarity

extracting for similarity search, Fine Tuning Without Fully Connected Layers

fine tuning, Improving Accuracy with Fine Tuning

Siamese networks and, Siamese Networks for One-Shot Face Verification

TensorFlow Embedding projector, Visualizing Image Clusters with t-SNE

visualizing, Improving Accuracy with Fine Tuning

encryption, Privacy

energy consumption, of mobile apps, Measuring Energy Impact-Benchmarking Load

episodes, Practical Deep Reinforcement Learning with DeepRacer

Euclidean distance, Similarity Search, Reducing Feature-Length with PCA

experience buffer, Practical Deep Reinforcement Learning with DeepRacer

experimental optimizations, Turn on Experimental Optimizations

explainability, Accountability and Explainability

tf-explain, tf-explain-tf-explain

What-If Tool, What-If Tool-What-If Tool

F

facial recognition, Siamese Networks for One-Shot Face Verification, Face Contours in ML Kit, Face Detection in Seeing AI

failure handling, Failure Handling

fairing, Fairing

Faiss, Faiss

Fast.ai, Education and Exploration

FastProgress, Training

Fatkun Batch Download Image, Data, Approach 2: Fatkun Chrome Browser Extension

feature extraction, Feature Extraction-Feature Extraction, Feature Extraction

feature scaling, Predicting an Image’s Category

feature vectors

defined, Image Similarity

length of, Length of Feature Vectors

reducing feature length, Reducing Feature-Length with PCA

federated learning, Federated Learning

Field Programmable Gate Arrays (FPGAs), FPGA + PYNQ

filter operations, Turn on Experimental Optimizations

filter_fusion option, Filter fusion

fine tuning, Fine Tuning-How Much to Fine Tune, Improving Accuracy with Fine Tuning-Fine Tuning Without Fully Connected Layers

fine-grained recognition, ImageNet Dataset

Firebase, ML Kit, ML Kit + Firebase

fixed round-robin order, Enable Nondeterministic Ordering

Flask, Flask: Build Your Own Server-Cons of Using Flask

Flickr, Flickr

food classifier apps

building real-time object recognition apps, Building a Real-Time Object Recognition App-Building a Real-Time Object Recognition App

challenges of, Shazam for Food: Developing Android Apps with TensorFlow Lite and ML Kit

life cycle of, The Life Cycle of a Food Classifier App

life cycle of mobile AI development, A Holistic Look at the Mobile AI App Development Cycle-How Do I Update the Model on My Users' Phones?

using ML Kit + Firebase, ML Kit + Firebase-Using the Experiment in Code

model conversion to TensorFlow Lite, Model Conversion to TensorFlow Lite

Shazam for Food app, Shazam for Food: Developing Android Apps with TensorFlow Lite and ML Kit

TensorFlow Lite architecture, TensorFlow Lite Architecture

TensorFlow Lite on iOS, TensorFlow Lite on iOS

TensorFlow Lite overview, An Overview of TensorFlow Lite

tools for, The Life Cycle of a Food Classifier App

forward pass, Batch Size

FPGAs (Field Programmable Gate Arrays), FPGA + PYNQ

Fritz, Fritz, Fritz-Fritz

fused operations, Use Fused Operations

G

GAN Lab, GAN Lab

General Data Protection Regulation (GDPR), Real-Time Object Classification on iOS with Core ML

generalization, Overtrain, and Then Generalize, Approach 2: Fatkun Chrome Browser Extension

Generative Adversarial Networks (GANs), pix2pix

geographic availability, Geographic Availability

Giphy, Giphy

Giraffe-Tree problem, Image Captioning

Google Cloud ML Engine

building classification APIs, Building a Classification API-Building a Classification API

pros and cons of, Pros of Using Cloud ML Engine

Google Cloud Platform (GCP), Uploading a hosted model

Google Cloud Vision, Google Cloud Vision, Invoking Prebuilt Cloud-Based Object Detection APIs

Google Coral USB accelerator, Google Coral USB Accelerator, Speeding Up with the Google Coral USB Accelerator

Google Dataset Search, Data

Google Edge Tensor Processing Unit (TPU), Google Coral USB Accelerator

Google Seedbank, Education and Exploration

Google's AI doodle, AI in the Browser with TensorFlow.js and ml5.js

GPU persistence, Enable GPU Persistence

GPU utilization, GPU Utilization

Graphics Processing Unit (GPU) starvation

displaying GPU statistics, nvidia-smi

overview of, GPU Starvation

visualizing program execution, TensorFlow Profiler + TensorBoard

GUI-based model training tools, A continuously evolving landscape

H

hardware

installing optimized stacks for, Install an Optimized Stack for the

Hardware

need for deep learning, Hardware

using better, Use Better Hardware

heatmaps, Class Activation Maps-Class Activation Maps, tf-explain, Heatmap visualization

high availability, High Availability

HomeCourt, HomeCourt

horizontal scaling, Distribute Training

hosted models

A/B testing, A/B Testing Hosted Models-Using the Experiment in Code

accessing, Accessing a hosted model

with Firebase, Hosted Models

uploading, Uploading a hosted model

hosting and serving (see inference serving)

Hyperas, Training

Hyperopt, Training

hyperparameters

automatic tuning, Tools of the Trade, Tools to Automate Tuning for Maximum Accuracy-AutoKeras, Autotune Parameter Values, Training

effect on accuracy, How Hyperparameters Affect Accuracy-Effect of Change in Aspect Ratio on Transfer Learning

I

IBM Watson Visual Recognition, IBM Watson Visual Recognition

image captioning, Image Captioning

image classification

building custom classifiers in Keras, Building a Custom Classifier in Keras with Transfer Learning-Analyzing the Results

category unawareness, ImageNet Dataset

class activation maps (heat maps), Class Activation Maps-Class Activation Maps

using CNNs, A Shallow Dive into Convolutional Neural Networks

desired properties of image classifiers, Desired Properties of an Image Classifier

fine-grained recognition, ImageNet Dataset

how images store information, Predicting an Image's Category

ImageNet dataset, ImageNet Dataset-ImageNet Dataset

using Keras, Predicting an Image's Category-Predicting an Image's Category

model zoos, Model Zoos

preprocessing images, Predicting an Image's Category

simple pipeline for, Predicting an Image's Category

image labeling APIs, Effect of Resizing on Image Labeling APIs, How Do Label My Data?

image saliency, Class Activation Maps

image segmentation, Image Segmentation

image similarity, Image Similarity-Image Similarity

image size, Performance Tuning for Cloud APIs

ImageN, Data

ImageNet dataset, Predicting an Image's Category, ImageNet Dataset-ImageNet Dataset

ImageNet Large Scale Visual Recognition Challenge (ILSVRC), How Deep Learning Became a Thing, ImageNet Dataset

ImageNet-Utils, Approach 1: Find or Collect a Dataset

Imagenette, Data

iMerit, Data

implicit bias, Bias

in-group/out-group bias, Bias

Inception architectures, Google Cloud Vision

InDro Robotics, InDro Robotics

inference serving

- desirable qualities for, Desirable Qualities in a Production-Level Serving System-Support for Multiple Machine Learning Libraries
- frameworks for, Frameworks
- frequently asked questions, Scalable Inference Serving on Cloud with TensorFlow Serving and KubeFlow
- Google Cloud ML Engine, Google Cloud ML Engine: A Managed Cloud AI Serving Stack-Building a Classification API
- KubeFlow, KubeFlow-Installation
- price versus performance, Price Versus Performance Considerations-Cost Analysis of Building Your Own Stack
- server building with Flask, Flask: Build Your Own Server-Cons of Using Flask
- TensorFlow Serving, TensorFlow Serving

tools, libraries, and cloud services, Landscape of Serving AI Predictions

inference time, Inference Time

Inference-as-a-Service, Google Cloud ML Engine: A Managed Cloud AI Serving Stack, Price Versus Performance Considerations

installation, Installation-Installation

InstaLooter, Data

instance retrieval, Building a Reverse Image Search Engine: Understanding Embeddings

instance-level segmentation, Instance-level segmentation

InstaSaber, InstaSaber + YoPuppet

Instruction Set Architecture (ISA), FPGAs

Intel Movidius Neural Compute Stick 2, Intel Movidius Neural Compute Stick

Interactive Telecommunications Program (ITP), PoseNet

interactivity, Real-Time Object Classification on iOS with Core ML

Intersection over Union (IoU), Intersection over Union

iOS models (see Core ML)

J

JavaScript-based ML libraries

ConvNetJS, ConvNetJS

history of, JavaScript-Based Machine Learning Libraries: A Brief History

Keras.js, Keras.js

ONNX.js, ONNX.js

TensorFlow.js, TensorFlow.js

JetBot, JetBot

Jetson Nano, NVIDIA Jetson Nano, Port to NVIDIA Jetson Nano

Jupyter Notebooks

machine learning examples, Education and Exploration

running interactively in browsers, Installation

using with KubeFlow, Fairing

K

Keras

benefits of, A continuously evolving landscape, Introducing Keras

callbacks in, Callbacks

conversion to Core ML, Conversion from Keras, Model Conversion Using Core ML Tools

custom classifiers using transfer learning, Building a Custom Classifier in Keras with Transfer Learning-Analyzing the Results

deploying models to Flask, Deploying a Keras Model to Flask

fine tuning object classifiers with, Approach 3: Fine Tuning Using Keras

history of, Keras

image classification pipeline, Predicting an Image's Category-Predicting an Image's Category

key function for feature extraction, Feature Extraction

model accuracy, Investigating the Model-ImageNet Dataset

pretrained models in, Model Zoos

Keras functions

decode_predictions, Predicting an Image's Category

ImageDataGenerator, Data Augmentation

preprocess_input, Predicting an Image's Category

Keras Tuner, Tools of the Trade, Keras Tuner-Keras Tuner, Training

Keras.js, Keras.js

keypoint detection, PoseNet

Knock Knock, Training

KubeFlow

fairing, Fairing

installation, Installation

introduction to, KubeFlow

pipelines, Pipelines

tools available, KubeFlow

Kubernetes, KubeFlow

L

labeling, Datasets, Data, How Do I Label My Data?, Labeling the Data

latency, Low Latency, Real-Time Object Classification on iOS with Core ML

layers, effect on accuracy, Effect of Number of Layers Fine-Tuned in Transfer Learning

lazy learning, Similarity Search

learning rate

effect on accuracy, Effect of Learning Rate

finding optimal, Find the Optimal Learning Rate

role of, Set Training Parameters

localization, Localization

Lose It!, Lose It!

loss function, Set Training Parameters, Siamese Networks for One-Shot Face Verification

low latency, Low Latency

M

machine learning (ML)

Apple's machine learning architecture, Apple's Machine Learning Architecture-ML Performance Primitives, Approach 2: Use Create ML

common techniques for, Common Techniques for Machine Learning Experimentation-Reproducible Experiments

definition of, What Is AI?

JavaScript-based ML libraries, JavaScript-Based Machine Learning Libraries: A Brief History-TensorFlow.js

overview of, Machine Learning

resources for learning about, Education and Exploration, Further Exploration

support for multiple ML libraries, Support for Multiple Machine Learning Libraries

terminology used in, Bias

Magic Sudoku, Magic Sudoku

magnitude-based weight pruning, Prune the Model

map operations, Turn on Experimental Optimizations

map_and_filter_fusion option, Map and filter fusion

map_func function, Parallelize I/O and Processing

map_fusion option, Map fusion

Markov decision process (MDP), The Markov decision process

masks, Segmentation

Mean Average Precision (mAP), Mean Average Precision

Metacar, Metacar

metric learning, Siamese Networks for One-Shot Face Verification

metrics, Set Training Parameters

MicroController Units (MCUs), TensorFlow Lite Architecture, Arduino

Microsoft Cognitive Services, Microsoft Cognitive Services, Invoking Prebuilt Cloud-Based Object Detection APIs

minibatches, Use Larger Batch Size

MirroredStrategy, Distribute Training

ML Kit

- benefits of, ML Kit
- common ML tasks supported by, ML Kit + Firebase
- cross-platform adaptability, TensorFlow Lite on iOS
- using custom trained TensorFlow Lite models, ML Kit + Firebase, Custom Models in ML Kit
- facial contours in, Face Contours in ML Kit
- using hosted models, Hosted Models-Using the Experiment in Code

object classification, Object Classification in ML Kit

ml5.js, ml5.js

MLPerf, Examine Industry Benchmarks

MMdnn, Model

MNIST dataset, A Glimmer of Hope

mobile-based AI

Apple's machine learning architecture, Apple's Machine Learning Architecture-ML Performance Primitives, Approach 2: Use Create ML

building real-time object recognition apps, Building a Real-Time Object Recognition App-Building a Real-Time Object Recognition App, Building a Real-Time Object Recognition App-Building a Real-Time Object Recognition App

case studies, Case Studies-InstaSaber + YoPuppet, Case Studies-Real-Time Video Segmentation in YouTube Stories

challenges of, Real-Time Object Classification on iOS with Core ML

conversion to Core ML, Conversion to Core ML

Core ML alternatives, Alternatives to Core ML

Core ML history, A Brief History of Core ML

development life cycle for, The Development Life Cycle for Artificial Intelligence on Mobile

dynamic model deployment, Dynamic Model Deployment

frequently asked questions, Real-Time Object Classification on iOS with Core ML, A Holistic Look at the Mobile AI App Development Cycle-How Do I Update the Model on My Users' Phones?

Fritz end-to-end solution for, Fritz-Fritz

measuring energy consumption, Measuring Energy Impact-

Benchmarking Load

on-device training, On-Device Training

performance analysis, Performance Analysis-Benchmarking Models on iPhones

reducing app size, Reducing App Size-Use Create ML

self-evolving model, The Self-Evolving Model

MobileNet, Training

model accuracy

improving TensorFlow performance, Inference-Enable GPU Persistence

transfer learning with Keras, Analyzing the Results-Analyzing the Results

model architecture, Model Architecture

model cards, Model

model compression, Prune the Model

model definition, Model Definition

model inference (TensorFlow)

fused operations, Use Fused Operations

GPU persistence, Enable GPU Persistence

pruning, Prune the Model

quantization techniques, Quantize the Model

selecting efficient models, Use an Efficient Model

model size, Model Size

model testing, Test the Model

model tips and tricks, Model

model training

analyzing with TensorFlow profiler, TensorFlow Profiler + TensorBoard

approaches to, Training Our Model-Approach 3: Fine Tuning Using Keras

browser-based AI, Training in the Browser-GPU Utilization

mobile-based AI, How Do I Train My Model?

optimizing in TensorFlow, Training-Examine Industry Benchmarks

transfer learning with Keras, Set Training Parameters-Start Training

web UI-based tools, Approach 1: Use Web UI-based Tools-Approach 1: Use Web UI-based Tools

model versioning, Model Versioning

model zoos, Model Zoos

model-free reinforcement learning, Model free versus model based

ModelDepot.io, Model

ModelZoo.co, Model

monitoring, Monitoring

Movidius Neural Compute Stick 2, Intel Movidius Neural Compute Stick

MS COCO (Microsoft Common Objects in COntext), Datasets, Image Captioning

MS Paint, Model

multiclass classification, Multiclass classification

multilayer neural networks, A Glimmer of Hope

multiple libraries support, Support for Multiple Machine Learning Libraries

multiples of eight, Find the Optimal Learning Rate

MultiWorkerMirroredStrategy, Distribute Training

Myriad VPU, Intel Movidius Neural Compute Stick

N

National Institute of Standards and Technology (NIST), A Glimmer of Hope

NavLab, A Glimmer of Hope, A Brief History of Autonomous Driving

nearest-neighbor approach

- efficacy of, Image Captioning
- scaling reverse image search using, Scaling Similarity Search with Approximate Nearest Neighbors-Faiss
- similarity search using, Similarity Search-Similarity Search

negative classes, Data, Approach 2: Fatkun Chrome Browser Extension

Neighborhood Graph and Tree (NGT), NGT

Netron, Model

Neural Architecture Search (NAS), Tools of the Trade, AutoKeras, Training

neural networks (see also Convolutional Neural Networks)

- data augmentation, Data Augmentation
- definition of, Exciting Beginnings
- matrix-matrix multiplication in, Quantize the Model
- multilayer, A Glimmer of Hope
- overview of, Neural Networks

purpose of validation sets, Organize the Data

resources for learning about, Further Reading

shortcomings of, Shortcoming of Neural Networks

New York Times, The New York Times, Scalability

NIST dataset, A Glimmer of Hope

NN-SVG, Model

Non-Maximum Suppression (NMS), Non-Maximum Suppression

nondeterministic ordering, Enable Nondeterministic Ordering

normalization, Predicting an Image's Category

Not Hotdog app

building iOS app, Building the iOS App

collecting data, Collecting Data-Approach 3: Web Scraper Using Bing Image Search API

model conversion using Core ML, Model Conversion Using Core ML Tools

model training, Training Our Model-Approach 3: Fine Tuning Using Keras

overview of steps, Not Hotdog on iOS with Core ML and Create ML

number of nines, High Availability

NVIDIA Data Loading Library (DALI) , tf.image built-in augmentations

NVIDIA GPU Persistence Daemon, Enable GPU Persistence

NVIDIA Graphics Processing Units (GPUs), Installation

NVIDIA Jetson Nano, NVIDIA Jetson Nano, Port to NVIDIA Jetson Nano

nvidia-smi (NVIDIA System Management Interface), nvidia-smi, Training

O

object classification (see also food classifier apps)

building iOS app, Building the iOS App

building real-time apps for, Building a Real-Time Object Recognition App-Building a Real-Time Object Recognition App

challenges of on mobile devices, Real-Time Object Classification on iOS with Core ML, Shazam for Food: Developing Android Apps with TensorFlow Lite and ML Kit

collecting data, Collecting Data-Approach 3: Web Scraper Using Bing Image Search API

in ML Kit, Object Classification in ML Kit

model conversion using Core ML, Model Conversion Using Core ML Tools

model training, Training Our Model-Approach 3: Fine Tuning Using Keras

overview of steps, Not Hotdog on iOS with Core ML and Create ML

real-time apps for, Building a Real-Time Object Recognition App-Building a Real-Time Object Recognition App

object detection

approaches to, Approaches to Object Detection

building custom detectors without code, Building a Custom Detector Without Any Code-Building a Custom Detector Without Any Code

case studies, Case Studies-Autonomous Cars

evolution of, The Evolution of Object Detection

image segmentation, Image Segmentation

inspecting and training models, Inspecting the Model-Training

key terms in, Key Terms in Object Detection-Non-Maximum Suppression

model conversion, Model Conversion

performance considerations, Performance Considerations

prebuilt cloud-based APIs, Invoking Prebuilt Cloud-Based Object Detection APIs-Invoking Prebuilt Cloud-Based Object Detection APIs

reusing pretrained models, Reusing a Pretrained Model-Deploying to a Device

TensorFlow Object Detection API, Using the TensorFlow Object Detection API to Build Custom Models-Preprocessing the Data

types of computer-vision tasks, Types of Computer-Vision Tasks-Approaches to Object Detection

object localization, Detection

object segmentation, Segmentation

OCR APIs, Effect of Compression on OCR APIs

OmniEarth, OmniEarth

one shot learning, Siamese Networks for One-Shot Face Verification

online resources

code examples, Using Code Examples

for this book, How to Contact Us

ONNX.js, ONNX.js

Open Images V4 , Data

Open Neural Network Exchange (ONNX), A continuously evolving landscape, Model, ONNX.js

optimizations (TensorFlow Lite)

model optimization toolkit, TensorFlow Model Optimization Toolkit

quantizing, Quantizing with TensorFlow Lite Converter

optimizations (TensorFlow) (see also TensorFlow performance checklist)

Automatic Mixed Precision, Use Automatic Mixed Precision

better hardware, Use Better Hardware

distributed training, Distribute Training

eager execution and tf.function, Use tf.function

experimental optimizations, Turn on Experimental Optimizations

industry benchmarks, Examine Industry Benchmarks

larger batch size, Use Larger Batch Size

multiples of eight, Use Multiples of Eight

optimized hardware stacks, Install an Optimized Stack for the Hardware

optimizing parallel CPU threads, Optimize the Number of Parallel CPU Threads

overtraining and generalization, Overtrain, and Then Generalize

optimizers

effect on accuracy, Effect of Optimizers

role of, Set Training Parameters

ordering, nondeterministic, Enable Nondeterministic Ordering

overfitting, Data Augmentation, Callbacks

overtraining, Early Stopping, Overtrain, and Then Generalize

O’Reilly’s Online Learning platform, Education and Exploration

P

p-hacking, Reproducibility

PapersWithCode.com, Model, Education and Exploration

parallel processing, Parallelize CPU Processing, Optimize the Number of Parallel CPU Threads

PCA (Principle Component Analysis), Reducing Feature-Length with PCA

perceptrons, Exciting Beginnings, The Cold and Dark Days, Perceptron

performance tuning, Performance Tuning for Cloud APIs-Effect of Resizing on OCR APIs (see also TensorFlow performance checklist)

Personally Identifiable Information (PII), Real-Time Object Classification on iOS with Core ML

Photobucket , Photobucket

Pilot Parliaments Benchmark (PPB), Bias

Pinterest, Pinterest

pipelines

- building in custom classifiers, Build the Data Pipeline
- end-to-end learning example pipeline, End-to-End Deep Learning Example Pipeline-Basic Custom Network Pipeline
- end-to-end object recognition example, Not Hotdog on iOS with Core ML and Create ML-Summary
- in KubeFlow, Pipelines
- reading data in TensorFlow, Use tf.data
- ready-to-use datasets for, TensorFlow Datasets
- simple pipeline for image classification, Predicting an Image’s Category

tuning TensorFlow performance, How to Use This Checklist-Enable GPU Persistence

pix2pix, pix2pix-pix2pix

Pixel 3 phones, Portrait Mode on Pixel 3 Phones

pixels, Predicting an Image's Category

PlotNeuralNet, Model

policy-based reinforcement learning, Policy based

pooling operations, Length of Feature Vectors, Pooling

portrait mode, Portrait Mode on Pixel 3 Phones

PoseNet, PoseNet, Squatting for Metro Tickets

pretrained models

adapting to new frameworks, Model

adapting to new tasks, Adapting Pretrained Models to New Tasks-How Much to Fine Tune

from Apple Machine Learning website, ML Performance Primitives

in Keras, Model Zoos

locating, Model

reusing for object detection, Reusing a Pretrained Model-Deploying to a Device

in TensorFlow.js, Running Pretrained Models Using TensorFlow.js

principal components, Reducing Feature-Length with PCA

privacy, Privacy, Privacy, Real-Time Object Classification on iOS with Core ML

profiling, TensorFlow Profiler + TensorBoard

progress bars, Training

progressive augmentation, Use progressive augmentation

progressive resizing, Use progressive resizing

progressive sampling, Use progressive sampling

Project Oxford, Microsoft Cognitive Services

proximal policy optimization (PPO) algorithm, Configure the simulation environment, Reinforcement Learning Algorithm in AWS DeepRacer

pruning, Prune the Model

PYNQ platform, FPGA + PYNQ

PyTorch, PyTorch

Q

Qri, Data

quantization techniques, Quantize the Model, Use Quantization, Quantizing with TensorFlow Lite Converter

Quilt, Data

R

randomization, Reproducible Experiments

Raspberry Pi, Raspberry Pi, Hands-On with the Raspberry Pi

region of interest (ROI), Identifying the Region of Interest

reinforcement learning

- AWS DeepRacer algorithm, Reinforcement Learning Algorithm in AWS DeepRacer
- AWS DeepRacer car, Racing the AWS DeepRacer Car-Sim2Real

transfer

AWS DeepRacer example, Practical Deep Reinforcement Learning with DeepRacer-Step 4: Evaluating the Performance of the Model

crux of, Building an Autonomous Car in Under an Hour: Reinforcement Learning with AWS DeepRacer

Distributed Deep Reinforcement Learning for Autonomous Driving tutorial , Reinforcement Learning

further exploration, Further Exploration-Roborace

improving learning models, Step 5: Improving Reinforcement Learning Models-Racing the AWS DeepRacer Car

inner workings of, Reinforcement Learning in Action

introduction to, A Brief Introduction to Reinforcement Learning

learning with autonomous cars, Why Learn Reinforcement Learning with an Autonomous Car?-Why Learn Reinforcement Learning with an Autonomous Car?

summary of, Deep Reinforcement Learning Summary with DeepRacer as an Example

terminology used in, Practical Deep Reinforcement Learning with DeepRacer

theory of, Reinforcement Learning Theory

replay buffer, Practical Deep Reinforcement Learning with DeepRacer

reporting bias, Bias

reproducibility, Reproducibility, Reproducible Experiments

ResearchCode, Education and Exploration

resizing

effect on accuracy, Effect of Resizing

images for performance tuning, Performance Tuning for Cloud APIs

ResNet, Training

ResNet-50, Predicting an Image's Category, Predicting an Image's Category

REST APIs, Making a REST API with Flask

reverse image search

case studies, Case Studies-Image Captioning

feature extraction, Feature Extraction-Feature Extraction

fine tuning for improved accuracy, Improving Accuracy with Fine Tuning-Fine Tuning Without Fully Connected Layers

image similarity, Image Similarity-Image Similarity

improving search speed, Improving the Speed of Similarity Search-Reducing Feature-Length with PCA

introduction to, Building a Reverse Image Search Engine: Understanding Embeddings

scaling with nearest-neighbor approach, Scaling Similarity Search witl Approximate Nearest Neighbors-Faiss

Siamese networks for, Siamese Networks for One-Shot Face Verification

similarity search, Similarity Search-Similarity Search

visualizing image clusters, Visualizing Image Clusters with t-SNE-Visualizing Image Clusters with t-SNE

reward function, Practical Deep Reinforcement Learning with DeepRacer, Configure reward function-Configure reward function, How Does a Reinforcement Learning System Learn?

Roborace, Roborace

robustness, Robustness

ROCm stack, Installation

root-mean-square error (RMSE), Set Training Parameters

round-robin order, Enable Nondeterministic Ordering

Runway ML, Education and Exploration

S

SageMaker reinforcement learning, A Brief Introduction to Reinforcement Learning

SamaSource, Data

scalability, Scalability

scaling horizontally, Distribute Training

ScrapeStorm.com, Data

Scrapy.org, Data

Seeing AI, Seeing AI, Face Detection in Seeing AI

segmentation, Segmentation

selection bias, Bias

self-driving cars, A Glimmer of Hope, Autonomous Cars (see also autonomous driving; reinforcement learning)

self-evolving model, The Self-Evolving Model

semantic segmentation, Semantic segmentation

Semi-Conductor, Semi-Conductor

semi-supervised learning, Approach 3: Web Scraper Using Bing Image Search API

sequential data, Training on Sequential Data

Shazam for Food app, Shazam for Food: Developing Android Apps with TensorFlow Lite and ML Kit

Siamese networks, Siamese Networks for One-Shot Face Verification

Sim2Real transfer, Sim2Real transfer

similarity search

- finding nearest neighbors, Similarity Search-Similarity Search
- improving speed of, Improving the Speed of Similarity Search-Reducing Feature-Length with PCA
- scaling, Scaling Similarity Search with Approximate Nearest Neighbors-Faiss

simulation, Deep Learning, Autonomous Driving, and the Data Problem, Practical Deep Reinforcement Learning with DeepRacer, Sim2Real transfer

simulation-to-real (sim2real) problem, Why Learn Reinforcement Learning with an Autonomous Car?, Practical Deep Reinforcement Learning with DeepRacer, Sim2Real transfer

smart refrigerators, Smart Refrigerator

softmax activation function, Start Training

SOTAWHAT tool, Education and Exploration

speaker recognition, Speaker Recognition by Alibaba

Spotify, Spotify

squat-tracker app, Squatting for Metro Tickets

SSD MobileNetV2, Reusing a Pretrained Model

Staples, Staples

stop conditions, Configure stop conditions

success, measuring, How Do I Measure the Success of My Model?

synthetic datasets, Which Library Should I Use?, Data

T

t-SNE algorithm, Visualizing Image Clusters with t-SNE-Visualizing Image Clusters with t-SNE

Teachable Machine, Training in the Browser

Tencent ML Images, Data

TensorBoard, TensorBoard-What-If Tool, TensorFlow Profiler + TensorBoard

TensorFlow

- AMD GPUs and, Installation
- benefits of, A continuously evolving landscape
- conversion to Core ML, Conversion from TensorFlow
- debugging scripts, Training
- history of, TensorFlow
- installation on Windows PCs, Installation
- testing for available GPUs, Training

TensorFlow Datasets, TensorFlow Datasets, Use TensorFlow Datasets, Data

TensorFlow Debugger (tfdbg), Training

TensorFlow Embedding projector, Visualizing Image Clusters with t-SNE

TensorFlow Encrypted, Privacy

TensorFlow Extended (TFX), TensorFlow Serving

TensorFlow Hub (tfhub.dev), Model

TensorFlow Lite

architecture, TensorFlow Lite Architecture

building real-time object recognition apps, Building a Real-Time Object Recognition App-Building a Real-Time Object Recognition App

history of, TensorFlow Lite

model conversion to, Model Conversion to TensorFlow Lite

model optimization toolkit, TensorFlow Model Optimization Toolkit

on iOS, TensorFlow Lite on iOS

overview of, An Overview of TensorFlow Lite

performance optimizations, Performance Optimizations

sample apps in repository, Building a Real-Time Object Recognition App

using models with ML Kit, ML Kit + Firebase, Custom Models in ML Kit

TensorFlow models repository, Building a Real-Time Object Recognition App, Obtaining the Model

TensorFlow Object Detection API

data collection, Data Collection

data labeling, Labeling the Data

data preprocessing, Preprocessing the Data

TensorFlow performance checklist

data augmentation, Data Augmentation-NVIDIA DALI

data preparation, Data Preparation-Use TensorFlow Datasets

data reading, Data Reading-Autotune Parameter Values

effective use of, How to Use This Checklist

GPU starvation, GPU Starvation-TensorFlow Profiler + TensorBoard

inference, Inference-Enable GPU Persistence

overview of, Data Preparation

training, Training-Examine Industry Benchmarks

TensorFlow Playground, TensorFlow.js

TensorFlow Serving, TensorFlow Serving

TensorFlow.js

architecture, TensorFlow.js Architecture

introduction to, TensorFlow.js

running pretrained models, Running Pretrained Models Using TensorFlow.js

TensorSpace, TensorSpace

tf-coreml, Conversion from TensorFlow

tf-explain, Tools of the Trade, tf-explain

tf.data, Use tf.data, Enable Nondeterministic Ordering

tf.data.experimental.AUTOTUNE, Autotune Parameter Values

tf.data.experimental.OptimizationOptions, Turn on Experimental Optimizations

tf.function, Use tf.function

tf.image, tf.image built-in augmentations

tf.keras, Tools of the Trade

tf.test.is_gpu_available(), Training

tfprof (TensorFlow profiler), TensorFlow Profiler + TensorBoard

TFRecords, Store as TFRecords, Preprocessing the Data

timeit command, Brute Force

tools, tips, and tricks

- data, Data-Data
- education and exploration, Education and Exploration-Education and Exploration
- installation, Installation-Installation
- models, Model
- privacy, Privacy
- training, Training

training parameters, Set Training Parameters

Transaction Processing Council (TPC) benchmark, Examine Industry Benchmarks

transfer learning

- adapting pretrained models to new networks, Model
- adapting pretrained models to new tasks, Adapting Pretrained Models to New Tasks-How Much to Fine Tune
- basic pipeline for, Basic Transfer Learning Pipeline
- building custom classifiers using, Building a Custom Classifier in Keras with Transfer Learning-Analyzing the Results
- Create ML, Approach 2: Use Create ML
- definition of, Datasets
- effect of hyperparmeters on accuracy, How Hyperparameters Affect Accuracy-Effect of Change in Aspect Ratio on Transfer Learning
- fine tuning, Fine Tuning-How Much to Fine Tune

GUI-based model training tools, A continuously evolving landscape

overview of, Transfer Learning

versus training from scratch, Transfer Learning Versus Training from Scratch

triplet loss function, Siamese Networks for One-Shot Face Verification

U

Uber, Uber

Ubuntu, Installation

underfitting, Data Augmentation

Uniform Manifold Approximation and Projection (UMAP), Visualizing Image Clusters with t-SNE

Universal Approximation Theorem, A Glimmer of Hope

updates, distributing, How Do I Update the Model on My Users' Phones?

USB accelerators, Google Coral USB Accelerator, Speeding Up with the Google Coral USB Accelerator

V

validation accuracy, Start Training

value function (V), Practical Deep Reinforcement Learning with DeepRacer

value-based reinforcement learning, Value based

vanilla policy gradient algorithm, Reinforcement Learning Algorithm in AWS DeepRacer

versioning, Training, Data, Model Versioning

video datasets, Data

video segmentation, Real-Time Video Segmentation in YouTube Stories

visual recognition APIs (see also computer vision)

accuracy, Accuracy

available APIs, The Landscape of Visual Recognition APIs-What's unique about this API?

bias, Bias-Bias

cost, Cost

service offerings, Service Offerings

VisualData.io, Data

visualizations

real-time for training process, Training

using GAN Lab, GAN Lab

using t-SNE algorithm, Visualizing Image Clusters with t-SNE-Visualizing Image Clusters with t-SNE

using TensorBoard, TensorBoard-What-If Tool

using TensorFlow profiler, TensorFlow Profiler + TensorBoard

using tf-explain, tf-explain

using What-If Tool, What-If Tool-What-If Tool

W

Waymo, Deep Learning, Autonomous Driving, and the Data Problem

web UI-based tools, Approach 1: Use Web UI-based Tools-Approach 1: Use Web UI-based Tools

WebScraper.io, Data

weights, Model Architecture

Weights and Biases, Training

What-If Tool, Tools of the Trade, What-If Tool-What-If Tool

wildlife conservation, Wildlife conservation

X

Xcode, Building a Real-Time Object Recognition App

Xilinx Zynq chips, FPGA + PYNQ

Y

YFCC100M, Data

YoPuppet, InstaSaber + YoPuppet

YouTube Stories, Real-Time Video Segmentation in YouTube Stories

Weights and Biases, Training

What-If Tool, Tools of the Trade, What-If Tool-What-If Tool

wildlife conservation, Wildlife conservation

X

Xcode, Building a Real-Time Object Recognition App

Xilinx Zynq chips, FPGA + PYNQ

Y

YFCC100M, Data

YoPuppet, InstaSaber + YoPuppet

YouTube Stories, Real-Time Video Segmentation in YouTube Stories

About the Authors
Primary Authors

Anirudh Koul is a noted AI expert, a UN/TEDx speaker, and a former senior scientist at Microsoft AI & Research, where he founded Seeing AI, often considered the most used technology among the blind community after the iPhone. Anirudh serves as the head of AI & Research at Aira, recognized by *Time* magazine as one of the best inventions of 2018. With features shipped to a billion users, he brings over a decade of production-oriented applied research experience on petabyte-scale datasets. He has been developing technologies using AI techniques for augmented reality, robotics, speech, and productivity as well as accessibility. His work with AI for Good, which IEEE has called "life-changing," has received awards from CES, FCC, MIT, Cannes Lions, and the American Council of the Blind; been showcased at events by UN, World Economic Forum, White House, House of Lords, Netflix, and National Geographic; and lauded by world leaders including Justin Trudeau and Theresa May.

Siddha Ganju, an AI researcher who *Forbes* featured in its "30 under 30" list, is a self-driving architect at Nvidia. As an AI advisor to NASA FDL, she helped build an automated meteor detection pipeline for the CAMS project at NASA, which discovered a comet. Previously at Deep Vision, she developed deep learning models for resource constraint edge devices. Her work ranges from visual question answering to generative adversarial networks to gathering insights from CERN's petabyte-scale data and has been published at top-tier conferences including CVPR and NeurIPS. She has served as a featured jury member in several international tech competitions including CES. As an advocate for diversity and inclusion in technology, she speaks at schools and colleges to motivate and grow a new generation of technologists from all backgrounds.

Meher Kasam is a seasoned software developer with apps used by tens of millions of users every day. Currently an iOS developer at Square, and having previously worked at Microsoft and Amazon, he has shipped features for a range of apps from Square's Point of Sale to the Bing iPhone app. During his time at Microsoft, he was the mobile development lead for the Seeing AI app, which has received widespread recognition and awards from Mobile World Congress, CES

FCC, and the American Council of the Blind, to name a few. A hacker at heart with a flair for fast prototyping, he’s won several hackathons and converted them to features shipped in widely used products. He also serves as a judge of international competitions including Global Mobile Awards and Edison Awards.

s

e

s

,

FCC, and the American Council of the Blind, to name a few. A hacker at heart with a flair for fast prototyping, he’s won several hackathons and converted them to features shipped in widely used products. He also serves as a judge of international competitions including Global Mobile Awards and Edison Awards.

Guest Authors

Chapter 17

Sunil Mallya is a principal deep learning scientist at AWS focused on deep learning and reinforcement learning at AWS. Sunil works with AWS customers in various transformation and innovation initiatives across verticals by building models for cutting edge machine learning applications. He also led the science development for AWS DeepRacer. Prior to joining AWS, Sunil cofounded the neuroscience and machine learning–based image analysis and video thumbnail recommendation company Neon Labs. He has worked on building large-scale low-latency systems at Zynga and has an acute passion for serverless computing He holds a master's degree in computer science from Brown University.

Chapter 16

Aditya Sharma is a program manager at Microsoft where he works on the Azure Autonomous Driving team. His work focuses on helping autonomous driving companies scale their operations by leveraging the power of the cloud, greatly reducing their time to market. He is the lead PM for Project Road Runner, the team behind the Autonomous Driving Cookbook. He also leads the Deep Learning and Robotics chapter at the Microsoft Garage. Aditya holds a graduate degree from the Robotics Institute at Carnegie Mellon University.

Chapter 16

Mitchell Spryn graduated from the University of Alabama with a dual bachelor's degree in electrical engineering and physics. While at the university, Mitchell was involved in a variety of research projects ranging from wireless power transfer to autonomous robotics. He currently works at Microsoft as a data scientist, specializing in distributed relational databases. In addition to his work on databases, Mitchell continues to contribute to projects in the robotics space such as the AirSim simulator and the Autonomous Driving Cookbook.

Chapter 15

Sam Sterckval, a Belgium-based electronics engineer, discovered his second passion (besides electronics) during his time in college—artificial intelligence. After graduation, Sam cofounded the company Edgise to combine these two passions—to bring artificial intelligence to the edge by designing custom hardware that can efficiently run complex AI models. Often considered the go-to person for edge AI in Belgium, Sam helps companies to build cognitive

solutions that highly depend on low latency, privacy, and cost-effectiveness.

Chapter 10

Zaid Alyafeai is known for his popular blogs and Google Colab Notebooks on TensorFlow.js and using deep learning within the browser to power interesting scenarios. He is also a writer for the official TensorFlow blog on Medium, with many GitHub projects with more than 1000 stars combined. When he is not building fancy web applications, he is conducting research in AI as a PhD student at KFUPM in Saudi Arabia. A teacher at heart, he also wrote a reference book on advanced integration techniques.

.

)

solutions that highly depend on low latency, privacy, and cost-effectiveness.

Chapter 10

Zaid Alyafeai is known for his popular blogs and Google Colab Notebooks on TensorFlow.js and using deep learning within the browser to power interesting scenarios. He is also a writer for the official TensorFlow blog on Medium, with many GitHub projects with more than 1000 stars combined. When he is not building fancy web applications, he is conducting research in AI as a PhD student at KFUPM in Saudi Arabia. A teacher at heart, he also wrote a reference book on advanced integration techniques.

Colophon

The animal on the cover of *Practical Deep Learning for Cloud, Mobile, and Edge* is the American peregrine falcon (*Falco peregrinus anatum*) or duck hawk a subspecies of peregrine falcon, which is currently found in the Rocky Mountains. This subspecies has become extinct in eastern North America, but healthy hybrid populations now exist due to organized reintroductions.

Peregrine falcons worldwide, no matter the subspecies, share the characteristics of being a large, fast-flying bird of prey with a dark head and wings; pale, patterned undersides; yellow beak and legs; and large eyes set in a distinctive vertical stripe of eyeblack. Falcons can be discerned at a distance by their "bent" but otherwise sharply outlined wings. Females are noticeably larger than males, but both average about the size of an American crow, and make a distinctive, sharp, "kak kak kak" call when defending territory or nestlings.

Peregrines (named for their far-wandering habits outside nesting season) have established themselves on all continents except Antarctica. Their species success lies with their many adaptations for hunting birds, as well as their ability to adap to varying nesting environments and prey. The best-known habit of the peregrine falcon is its hunting stoop or high-speed dive, in which it dives from a great height into a bird on or near the ground, reaching speeds of up to 200 mph by the time it crashes into its prey, many times killing it instantly. This makes it not only the fastest bird on earth, but the fastest member of the animal kingdom, and is why the peregrine is a favorite bird of falconers around the world.

Peregrine falcons became a visible symbol of the US environmental movement in the 1960s, as widespread use of the pesticide DDT extirpated the birds across much of their former range before the chemical was banned (the birds consumed DDT through their prey, which had the effect of lethally thinning the shells of the falcon's eggs). The US Environmental Protection Agency declared this falcon an endangered species in 1970. However, the species has recovered due to the ban on DDT, followed by a captive breeding program that reintroduced the species into their former range in the eastern US. With nest boxes added into the upper reaches of city skyscrapers, populations quickly became established, as the falcons raised their chicks on pigeons and other plentiful urban birds. The peregrine falcon was removed from the Endangered Species list in 1999. While the American peregrine falcon's current conservation status is now designated as

of Least Concern, many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *British Birds*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.

of Least Concern, many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *British Birds*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.

1. Preface
    a. To the Backend/Frontend/Mobile Software Developer
    b. To the Data Scientist
    c. To the Student
    d. To the Teacher
    e. To the Robotics Enthusiast
    f. What to Expect in Each Chapter
    g. Conventions Used in This Book
    h. Using Code Examples
    i. O’Reilly Online Learning
    j. How to Contact Us
    k. Acknowledgments
        i. Group Acknowledgments
        ii. Personal Acknowledgments
2. 1. Exploring the Landscape of Artificial Intelligence
    a. An Apology
    b. The Real Introduction
    c. What Is AI?
        i. Motivating Examples
    d. A Brief History of AI
        i. Exciting Beginnings
        ii. The Cold and Dark Days
        iii. A Glimmer of Hope

    iv. How Deep Learning Became a Thing
  e. Recipe for the Perfect Deep Learning Solution
    i. Datasets
    ii. Model Architecture
    iii. Frameworks
    iv. Hardware
  f. Responsible AI
    i. Bias
    ii. Accountability and Explainability
    iii. Reproducibility
    iv. Robustness
    v. Privacy
  g. Summary
  h. Frequently Asked Questions
3. 2. What’s in the Picture: Image Classification with Keras
  a. Introducing Keras
  b. Predicting an Image’s Category
  c. Investigating the Model
    i. ImageNet Dataset
    ii. Model Zoos
    iii. Class Activation Maps
  d. Summary
4. 3. Cats Versus Dogs: Transfer Learning in 30 Lines with Keras

a. Adapting Pretrained Models to New Tasks
    i. A Shallow Dive into Convolutional Neural Networks
    ii. Transfer Learning
    iii. Fine Tuning
    iv. How Much to Fine Tune
b. Building a Custom Classifier in Keras with Transfer Learning
c. Organize the Data
d. Build the Data Pipeline
    i. Number of Classes
    ii. Batch Size
e. Data Augmentation
f. Model Definition
g. Train the Model
    i. Set Training Parameters
    ii. Start Training
h. Test the Model
i. Analyzing the Results
j. Further Reading
k. Summary

5. 4. Building a Reverse Image Search Engine: Understanding Embeddings
    a. Image Similarity
    b. Feature Extraction
    c. Similarity Search

d. Visualizing Image Clusters with t-SNE
e. Improving the Speed of Similarity Search
    i. Length of Feature Vectors
    ii. Reducing Feature-Length with PCA
f. Scaling Similarity Search with Approximate Nearest Neighbors
    i. Approximate Nearest-Neighbor Benchmark
    ii. Which Library Should I Use?
    iii. Creating a Synthetic Dataset
    iv. Brute Force
    v. Annoy
    vi. NGT
    vii. Faiss
g. Improving Accuracy with Fine Tuning
    i. Fine Tuning Without Fully Connected Layers
h. Siamese Networks for One-Shot Face Verification
i. Case Studies
    i. Flickr
    ii. Pinterest
    iii. Celebrity Doppelgangers
    iv. Spotify
    v. Image Captioning
j. Summary

6. 5. From Novice to Master Predictor: Maximizing Convolutional Neural

Network Accuracy

a. Tools of the Trade
    i. TensorFlow Datasets
    ii. TensorBoard
    iii. What-If Tool
    iv. tf-explain
b. Common Techniques for Machine Learning Experimentation
    i. Data Inspection
    ii. Breaking the Data: Train, Validation, Test
    iii. Early Stopping
    iv. Reproducible Experiments
c. End-to-End Deep Learning Example Pipeline
    i. Basic Transfer Learning Pipeline
    ii. Basic Custom Network Pipeline
d. How Hyperparameters Affect Accuracy
    i. Transfer Learning Versus Training from Scratch
    ii. Effect of Number of Layers Fine-Tuned in Transfer Learning
    iii. Effect of Data Size on Transfer Learning
    iv. Effect of Learning Rate
    v. Effect of Optimizers
    vi. Effect of Batch Size
    vii. Effect of Resizing
    viii. Effect of Change in Aspect Ratio on Transfer Learning

e. Tools to Automate Tuning for Maximum Accuracy
    i. Keras Tuner
    ii. AutoAugment
    iii. AutoKeras
f. Summary

7. 6. Maximizing Speed and Performance of TensorFlow: A Handy Checklist
    a. GPU Starvation
        i. nvidia-smi
        ii. TensorFlow Profiler + TensorBoard
    b. How to Use This Checklist
    c. Performance Checklist
        i. Data Preparation
        ii. Data Reading
        iii. Data Augmentation
        iv. Training
        v. Inference
    d. Data Preparation
        i. Store as TFRecords
        ii. Reduce Size of Input Data
        iii. Use TensorFlow Datasets
    e. Data Reading
        i. Use tf.data
        ii. Prefetch Data

iii. Parallelize CPU Processing
iv. Parallelize I/O and Processing
v. Enable Nondeterministic Ordering
vi. Cache Data
vii. Turn on Experimental Optimizations
viii. Autotune Parameter Values

f. Data Augmentation
    i. Use GPU for Augmentation

g. Training
    i. Use Automatic Mixed Precision
    ii. Use Larger Batch Size
    iii. Use Multiples of Eight
    iv. Find the Optimal Learning Rate
    v. Use tf.function
    vi. Overtrain, and Then Generalize
    vii. Install an Optimized Stack for the Hardware
    viii. Optimize the Number of Parallel CPU Threads
    ix. Use Better Hardware
    x. Distribute Training
    xi. Examine Industry Benchmarks

h. Inference
    i. Use an Efficient Model
    ii. Quantize the Model

iii. Prune the Model

iv. Use Fused Operations

v. Enable GPU Persistence

i. Summary

8. 7. Practical Tools, Tips, and Tricks

a. Installation

b. Training

c. Model

d. Data

e. Privacy

f. Education and Exploration

g. One Last Question

9. 8. Cloud APIs for Computer Vision: Up and Running in 15 Minutes

a. The Landscape of Visual Recognition APIs

i. Clarifai

ii. Microsoft Cognitive Services

iii. Google Cloud Vision

iv. Amazon Rekognition

v. IBM Watson Visual Recognition

vi. Algorithmia

b. Comparing Visual Recognition APIs

i. Service Offerings

ii. Cost

iii. Accuracy

iv. Bias

c. Getting Up and Running with Cloud APIs

d. Training Our Own Custom Classifier

i. Top Reasons Why Our Classifier Does Not Work Satisfactorily

e. Comparing Custom Classification APIs

f. Performance Tuning for Cloud APIs

i. Effect of Resizing on Image Labeling APIs

ii. Effect of Compression on Image Labeling APIs

iii. Effect of Compression on OCR APIs

iv. Effect of Resizing on OCR APIs

g. Case Studies

i. The New York Times

ii. Uber

iii. Giphy

iv. OmniEarth

v. Photobucket

vi. Staples

vii. InDro Robotics

h. Summary

10. 9. Scalable Inference Serving on Cloud with TensorFlow Serving and KubeFlow

a. Landscape of Serving AI Predictions

b. Flask: Build Your Own Server
    i. Making a REST API with Flask
    ii. Deploying a Keras Model to Flask
    iii. Pros of Using Flask
    iv. Cons of Using Flask
c. Desirable Qualities in a Production-Level Serving System
    i. High Availability
    ii. Scalability
    iii. Low Latency
    iv. Geographic Availability
    v. Failure Handling
    vi. Monitoring
    vii. Model Versioning
    viii. A/B Testing
    ix. Support for Multiple Machine Learning Libraries
d. Google Cloud ML Engine: A Managed Cloud AI Serving Stack
    i. Pros of Using Cloud ML Engine
    ii. Cons of Using Cloud ML Engine
    iii. Building a Classification API
e. TensorFlow Serving
    i. Installation
f. KubeFlow
    i. Pipelines

    ii. Fairing
    iii. Installation
  g. Price Versus Performance Considerations
    i. Cost Analysis of Inference-as-a-Service
    ii. Cost Analysis of Building Your Own Stack
  h. Summary
11. 10. AI in the Browser with TensorFlow.js and ml5.js
  a. JavaScript-Based Machine Learning Libraries: A Brief History
    i. ConvNetJS
    ii. Keras.js
    iii. ONNX.js
    iv. TensorFlow.js
  b. TensorFlow.js Architecture
  c. Running Pretrained Models Using TensorFlow.js
  d. Model Conversion for the Browser
  e. Training in the Browser
    i. Feature Extraction
    ii. Data Collection
    iii. Training
    iv. GPU Utilization
  f. ml5.js
  g. PoseNet
  h. pix2pix

i. Benchmarking and Practical Considerations

i. Model Size

ii. Inference Time

j. Case Studies

i. Semi-Conductor

ii. TensorSpace

iii. Metacar

iv. Airbnb's Photo Classification

v. GAN Lab

k. Summary

12. 11. Real-Time Object Classification on iOS with Core ML

a. The Development Life Cycle for Artificial Intelligence on Mobile

b. A Brief History of Core ML

c. Alternatives to Core ML

i. TensorFlow Lite

ii. ML Kit

iii. Fritz

d. Apple's Machine Learning Architecture

i. Domain-Based Frameworks

ii. ML Framework

iii. ML Performance Primitives

e. Building a Real-Time Object Recognition App

f. Conversion to Core ML

    i. Conversion from Keras
    ii. Conversion from TensorFlow
  g. Dynamic Model Deployment
  h. On-Device Training
    i. Federated Learning
  i. Performance Analysis
    i. Benchmarking Models on iPhones
  j. Measuring Energy Impact
    i. Benchmarking Load
  k. Reducing App Size
    i. Avoid Bundling the Model
    ii. Use Quantization
    iii. Use Create ML
  l. Case Studies
    i. Magic Sudoku
    ii. Seeing AI
    iii. HomeCourt
    iv. InstaSaber + YoPuppet
  m. Summary
13. 12. Not Hotdog on iOS with Core ML and Create ML
  a. Collecting Data
    i. Approach 1: Find or Collect a Dataset
    ii. Approach 2: Fatkun Chrome Browser Extension

iii. Approach 3: Web Scraper Using Bing Image Search API

b. Training Our Model

i. Approach 1: Use Web UI-based Tools

ii. Approach 2: Use Create ML

iii. Approach 3: Fine Tuning Using Keras

c. Model Conversion Using Core ML Tools

d. Building the iOS App

e. Further Exploration

f. Summary

14. 13. Shazam for Food: Developing Android Apps with TensorFlow Lite and ML Kit

a. The Life Cycle of a Food Classifier App

b. An Overview of TensorFlow Lite

i. TensorFlow Lite Architecture

c. Model Conversion to TensorFlow Lite

d. Building a Real-Time Object Recognition App

e. ML Kit + Firebase

i. Object Classification in ML Kit

ii. Custom Models in ML Kit

iii. Hosted Models

iv. A/B Testing Hosted Models

v. Using the Experiment in Code

f. TensorFlow Lite on iOS

g. Performance Optimizations
    i. Quantizing with TensorFlow Lite Converter
    ii. TensorFlow Model Optimization Toolkit
h. Fritz
i. A Holistic Look at the Mobile AI App Development Cycle
    i. How Do I Collect Initial Data?
    ii. How Do I Label My Data?
    iii. How Do I Train My Model?
    iv. How Do I Convert the Model to a Mobile-Friendly Format?
    v. How Do I Make my Model Performant?
    vi. How Do I Build a Great UX for My Users?
    vii. How Do I Make the Model Available to My Users?
    viii. How Do I Measure the Success of My Model?
    ix. How Do I Improve My Model?
    x. How Do I Update the Model on My Users' Phones?
j. The Self-Evolving Model
k. Case Studies
    i. Lose It!
    ii. Portrait Mode on Pixel 3 Phones
    iii. Speaker Recognition by Alibaba
    iv. Face Contours in ML Kit
    v. Real-Time Video Segmentation in YouTube Stories
l. Summary

15. 14. Building the Purrfect Cat Locator App with TensorFlow Object Detection API
    a. Types of Computer-Vision Tasks
        i. Classification
        ii. Localization
        iii. Detection
        iv. Segmentation
    b. Approaches to Object Detection
    c. Invoking Prebuilt Cloud-Based Object Detection APIs
    d. Reusing a Pretrained Model
        i. Obtaining the Model
        ii. Test Driving Our Model
        iii. Deploying to a Device
    e. Building a Custom Detector Without Any Code
    f. The Evolution of Object Detection
        i. Performance Considerations
    g. Key Terms in Object Detection
        i. Intersection over Union
        ii. Mean Average Precision
        iii. Non-Maximum Suppression
    h. Using the TensorFlow Object Detection API to Build Custom Models
        i. Data Collection
        ii. Labeling the Data

iii. Preprocessing the Data

i. Inspecting the Model

i. Training

ii. Model Conversion

j. Image Segmentation

k. Case Studies

i. Smart Refrigerator

ii. Crowd Counting

iii. Face Detection in Seeing AI

iv. Autonomous Cars

l. Summary

16. 15. Becoming a Maker: Exploring Embedded AI at the Edge

a. Exploring the Landscape of Embedded AI Devices

i. Raspberry Pi

ii. Intel Movidius Neural Compute Stick

iii. Google Coral USB Accelerator

iv. NVIDIA Jetson Nano

v. FPGA + PYNQ

vi. Arduino

b. A Qualitative Comparison of Embedded AI Devices

c. Hands-On with the Raspberry Pi

d. Speeding Up with the Google Coral USB Accelerator

e. Port to NVIDIA Jetson Nano

f. Comparing the Performance of Edge Devices

g. Case Studies

    i. JetBot

    ii. Squatting for Metro Tickets

    iii. Cucumber Sorter

h. Further Exploration

i. Summary

17. 16. Simulating a Self-Driving Car Using End-to-End Deep Learning with Keras

    a. A Brief History of Autonomous Driving

    b. Deep Learning, Autonomous Driving, and the Data Problem

    c. The “Hello, World!” of Autonomous Driving: Steering Through a Simulated Environment

        i. Setup and Requirements

    d. Data Exploration and Preparation

        i. Identifying the Region of Interest

        ii. Data Augmentation

        iii. Dataset Imbalance and Driving Strategies

    e. Training Our Autonomous Driving Model

        i. Drive Data Generator

        ii. Model Definition

    f. Deploying Our Autonomous Driving Model

    g. Further Exploration

        i. Expanding Our Dataset

ii. Training on Sequential Data

iii. Reinforcement Learning

h. Summary

18. 17. Building an Autonomous Car in Under an Hour: Reinforcement Learning with AWS DeepRacer

a. A Brief Introduction to Reinforcement Learning

b. Why Learn Reinforcement Learning with an Autonomous Car?

c. Practical Deep Reinforcement Learning with DeepRacer

i. Building Our First Reinforcement Learning

ii. Step 1: Create Model

iii. Step 2: Configure Training

iv. Step 3: Model Training

v. Step 4: Evaluating the Performance of the Model

d. Reinforcement Learning in Action

i. How Does a Reinforcement Learning System Learn?

ii. Reinforcement Learning Theory

iii. Reinforcement Learning Algorithm in AWS DeepRacer

iv. Deep Reinforcement Learning Summary with DeepRacer as an Example

v. Step 5: Improving Reinforcement Learning Models

e. Racing the AWS DeepRacer Car

i. Building the Track

ii. AWS DeepRacer Single-Turn Track Template

iii. Running the Model on AWS DeepRacer

iv. Driving the AWS DeepRacer Vehicle Autonomously

f. Further Exploration

i. DeepRacer League

ii. Advanced AWS DeepRacer

iii. AI Driving Olympics

iv. DIY Robocars

v. Roborace

g. Summary

19. A. A Crash Course in Convolutional Neural Networks

a. Machine Learning

b. Perceptron

c. Activation Functions

d. Neural Networks

e. Backpropagation

f. Shortcoming of Neural Networks

g. Desired Properties of an Image Classifier

h. Convolution

i. Pooling

j. Structure of a CNN

k. Further Exploration

20. Index